НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет Інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:

Завідувач кафедри

_____ Сергій СТІРЕНКО

«___»_____20__ р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інженерія програмного забезпечення комп'ютерних систем»

спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Система виконання коду без використання методів збирання сміття»

Виконав:
студент IV курсу, групи ІП-64
Пархоменко Денис Володимирович _____

Керівник:
Старший викладач
Сімоненко Андрій Валерійович _____

Консультант з нормоконтролю:
Доктор технічних наук, професор
Сімоненко Валерій Павлович _____

Рецензент:
Посада, науковий ступінь, вчене звання
Призвище, ім'я, по батькові _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2020 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет Інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення комп'ютерних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Сергій СТІРЕНКО

«___»_____20__ р.

ЗАВДАННЯ

на дипломний проєкт студенту

Пархоменку Денису Володимировичу

1. Тема проєкту «Система виконання коду без використання методів збирання сміття», керівник проєкту старший викладач Сімоненко А. В., затверджені наказом по університету від «___»_____ 20__ р. №_____

2. Термін подання студентом проєкту 06.06.2020

3. Вихідні дані до проєкту технічна документація

4. Зміст пояснювальної записки:

Розділ 1. Огляд існуючих рішень

Розділ 2. Огляд архітектури системи

Розділ 3. Опис програмної реалізації

Розділ 4. Аналіз експерементальної реалізації

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо):

Схема 1. Структурна схема

Схема 2. Функціональна схема

Схема 3. Блок схема алгоритму

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Розділ 1	Сімоненко В. П., проф.		
Розділ 2	Сімоненко В. П., проф.		
Розділ 3	Сімоненко В. П., проф.		
Розділ 4	Сімоненко В. П., проф.		

7. Дата видачі завдання 09.12.2019

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Затвердження теми роботи	09.12.2019	
2.	Вивчення та аналіз завдання	15.12.2019 - 03.02.2020	
3.	Розробка архітектури та загальної структури систем	03.02.2020 - 16.03.2020	
4.	Розробка структур окремих компонентів системи	16.04.2020 - 11.04.2020	
5.	Програмна реалізація системи	11.04.2020 - 03.05.2020	
6.	Виправлення помилок	03.05.2020 - 08.052020	
7.	Оформлення пояснювальної записки	08.05.2020 - 06.06.2020	
8.	Передзахист		
9.	Захист		

Студент Денис, ПАРХОМЕНКО

Керівник Андрій, СІМОНЕНКО

АНОТАЦІЯ

У даній роботі детально розглянуту структуру основних типів систем виконання коду: стекових та регістрових. Розглянуті основні способи автоматичного управління пам’яттю, що використовуються в сучасних системах виконання. Запропоновано новий метод автоматичного управління пам’яттю оснований на циклах життя об’єктів системи. Було реалізовано віртуальну машину прикладного рівня із спеціально спроектованим байт-кодом, що реалізує новий механізм управління пам’яттю.

ABSTRACT

This work analyses in detail the structure of the main types of code execution systems: stack-based and register-based. It looks at the main methods of automatic memory management used in modern code execution systems. A new method of automatic memory management based on the lifecycles of system objects is proposed. An application-level virtual machine with a specially designed bytecode was implemented, that uses this new memory management mechanism.

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЄКТУ

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1	А4		Завдання на дипломний проєкт	2	
2	А4	ДП 6418. 01.000.ВП	Відомість дипломного проєкту	1	
3	А4	ДП 6418. 02.000.ТЗ	Технічне завдання	3	
4	А4	ДП 6418. 03.000.ПЗ	Пояснювальна записка	77	
5	А4	ДП 6418. 04.000.Д1	Структурна схема	1	
6	А4	ДП 6418. 05.000.Д2	Функціональна схема	1	
7	А4	ДП 6418. 06.000.Д3	Блок схема алгоритму	1	

				ДП 6418. 01.000.ВП		
	ПІБ	Підп.	Дата			
Розробн.	Пархоменко Д. В.			Система виконання код без використання методів збирання сміття	Лист	Листів
Керівн.	Сімоненко А.В				1	1
					КПІ ім. Ігоря Сікорського Каф. ОТ Гр. ІП-64	
Н/контр.	Сімоненко А.В					
Зав.каф.	Стіренко С.Г			Відомість дипломного проєкту		

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЕКТУ

на тему: «Система виконання коду без використання методів збирання сміття»

Київ – 2020 року

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання стосується розробки системи виконання коду, яка замість засобів збирання сміття використовує детермінований, але автоматичний спосіб управління пам’яттю.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання бакалаврської дипломної роботи, затверджене кафедрою обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут» імені Ігоря Сікорського.

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою розробки є створення системи виконання коду без використання методів збирання сміття.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки є науково-технічна література, монографії, публікації в періодичних виданнях та мережі Інтернет.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до програмного продукту, що розробляється

Система, що розробляється, повинна:

1) Використовувати детермінований, але автоматичний спосіб управління пам’яттю

2) виконувати спеціально розроблений байткод;
3) надавати можливість виконувати код на різних архітектурах та операційних системах.

5.2. Вимоги до інструментального програмного забезпечення

– середовище з встановленим Rustup або компілятором Rust nightly;

5.3. Вимоги до апаратної частини обчислювальної системи

– процесор з тактовою частотою не менше ніж 500 КГц;

– оперативна пам’ять об’ємом не менше ніж 128 Мб;

– жорсткий диск об’ємом не менше ніж 20 Мб.

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЕКТУ

на тему: «Система виконання коду без використання методів збирання сміття»

Київ – 2020

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕТЬ

VM – Віртуальна машина прикладного рівня

JVM – Віртуальна машина Java

CLR – Common language runtime

JIT – Just in time compilation

ОС – Операційна система

РС – Програмний покажчик

GC – Збирач сміття

RC – Підрахунок посилань

ВСТУП

Технології розвиваються та змінюються досить швидко. З появою інтернету речей з'явилася необхідність в програмах, що використовують менше ресурсів процесора. Але з точки зору зручності програміста заснований мовою Java принцип «Напиши раз, запускай усюди». Він основаним на побудові незалежних від архітектури процесора систем виконання коду.

Цей принцип заполонив світ розробки програмного забезпечення упродовж останніх 20 років, адже цей принцип дозволяє не переживати програмісту за управління пам'яттю та дозволяє написати справді апаратно незалежний код. Але існуючі системи виконання були розроблені досить давно, та приблизно в один і той же час, тому між ними є багато чого спільного: зокрема управління пам'яттю виконується одним із методів збирання сміття. Це призводить до недетермінованого стану пам'яті в певний момент, що може бути досить проблематично для систем з обмеженою пам'яттю. Крім того, ці системи були розробленні до існування 64 бітних процесорів та не враховують сучасні можливості процесорів у своїй архітектурі. Звісно можна писати на компільованих мовах програмування. Але в такому випадку програмісту доведеться самому відповідати за управління пам'яттю та за апаратну незалежність програм.

З урахуванням цього, у світі з'явилася ніша програмних задач, які потребують систем виконання коду з автоматичним, але детермінованим управлінням пам'яті. Така розробка була б корисною не лише у світі інтернету речей, а також у скриптових двигунах ігор та як універсальна система виконання коду для будь-яких задач.

Виходячи з вищесказаного, дана робота представляє саме таку систему, яка використовує детермінований метод управління пам'яттю, а також враховує архітектурні особливості сучасних процесорів, та розробляється саме під них, але може використовуватися для великого спектру апаратних архітектур.

РОЗДІЛ 1
АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

Нинішні реалізації систем виконання коду досить схожі одна на іншу, але між цим вони загалом орієнтовані на специфічну мову програмування та використовують або трасування, або RC для управління пам’яттю. Але все ж крім управління пам’яттю, VM складається із багатьох компонентів. Далі розглянемо основні компоненти найбільш популярних та сучасних віртуальних машин. JVM, та її реалізації є дуже добре задокументовані, тому почнемо із неї.

1.1 Java Virtual machine

Java Virtual machine – це віртуальна машина прикладного рівня, що запускає програми написані на мові Java. JVM має детальну специфікацію, яка описує поведінку JVM. Завдяки цій специфікації програміст може не переживати про деталі специфічної реалізації JVM чи апаратної платформи. Основною реалізації JVM є Hotspot JVM, що є частиною проекту OpenJDK з відкритим кодом та вбудованим JIT компілятором.

1.1.1 Принцип роботи Hotspot JVM

Розглянемо принцип роботи Hotspot JVM. Схема роботи JVM наведена на рисунку 1.1[1].

Hotspot JVM логічно розділена на 2 частини: компоненти, що створюються окремо для кожного потоку (JVM реалізує підтримку потоків на рівні VM) та компоненти незалежні від потоків [2].

1.1.1.1 Компоненти потоку

Потік в JVM – це одна логічна одиниця виконання програми. JVM дозволяє програмі одночасно виконувати декілька потоків. В JVM 1 потік прямо відображає нативний потік ОС. Після підготовки всього стану JVM потоку, створюється ОС потік. ОС несе відповідальність за планування всіх

потоків. Коли потік завершує роботу, то всі ресурси що буди їм зайняті, як зі сторони ОС, так і зі сторони JVM звільнюються.

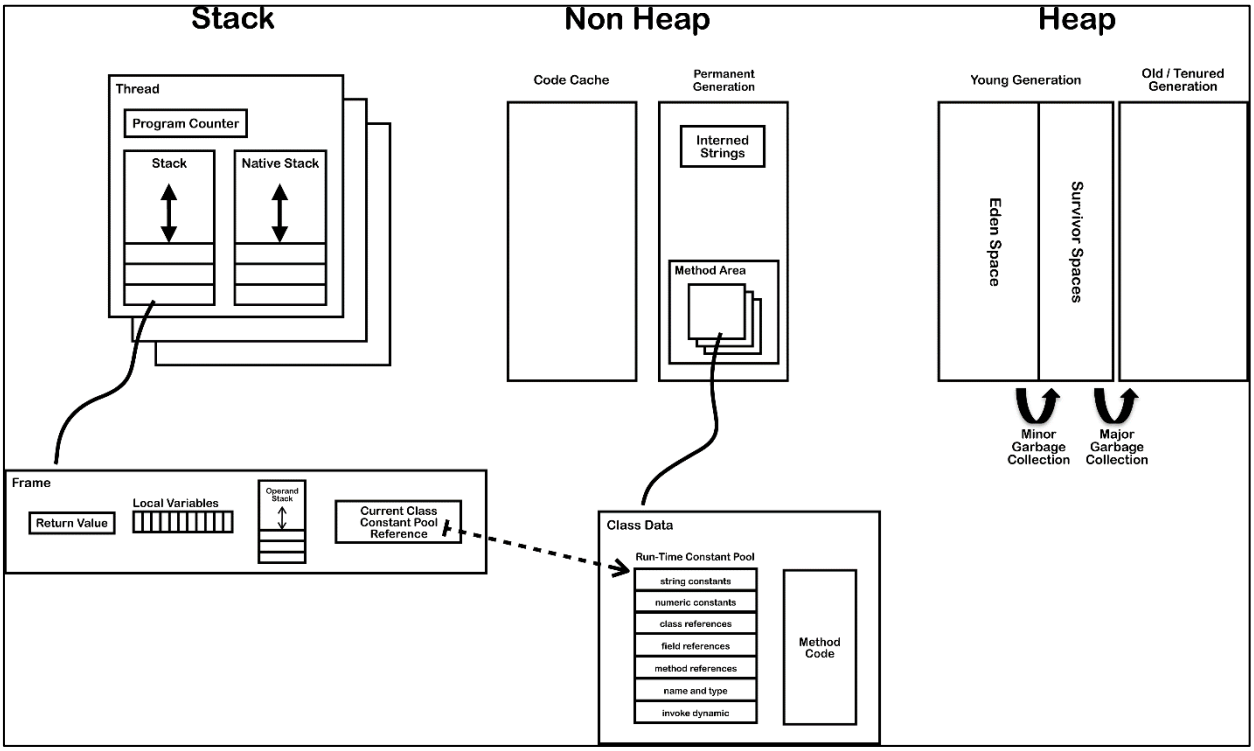


Рис. 1.1 – Схема роботи JVM

Окрім потоків, що виконують код користувача в JVM присутні декілька службових потоків, які виконують системні функції. Список даних потоків наведений в таблиці 1.1[2].

Таблиця 1.1 – Опис потоків JVM

Назва потоку	Опис
VM thread	Це головний потік програми VM. Окрім запуску програми, цей потік виконує операції які потребують знаходження VM в стані safe-point. Такі операції потребують стану JVM при якому модифікації Heap області гарантовано не відбудуться. Основною такою операцією славнозвісне трасування “stop-the-world”.

Таблиця 1.1 (Закінчення)

Назва потоку	Опис
Periodic task thread	Цей потік відповідає за переривання, що використовуються для планування виконання періодичних операцій.
GC threads	Дані потоки (їх може буди декілька), відповідають за збирання сміття в JVM, зазвичай одним із методів трасування.
Compiler threads	Ці потоки займаються JIT компіляцією байткоду в нативний код
Signal dispatcher thread	Цей потік обробляє сингали, що поступають процесу VM від ОС та викликає методи обробки сигналів в JVM

Кожний потік користувача має програмний покажчик. Він оновлюється після обробки команди, тому вказує на наступну команду. Якщо потік виконує нативний код, то значення PC не є визначеним. PC зберігається в пам’яті як покажчик та вказує на якусь адресу в Method area Vm.

Стек JVM це LIFO структура даних, зверху якої зберігаються дані про метод, що зараз виконується. Виклик кожного методу супроводжується створенням нового фрейму на стеці. Даний фрейм знищується коли код виходить із методу. Так як прямі маніпуляції із стеком не допустимі, то кожен стек фрейм може знаходитися в окремій частині Heap програми. Стек зазвичай є статичного розміру, але HotSpot також підтримує динамічний стек. Кожен потік JVM також підтримує нативний стек, який реалізований як звичайний C стек для ОС в якій виконується код [2].

Фрейм JVM зберігає дані про локальні змінні, значення яке повертається із методу та посилання на пул констант для класу поточного методу. Локальні змінні в свою чергу зберігаються в обмеженому масиві з яким власне може працювати байт-код. Локальні зміні можуть буть наступних типів:

– boolean
– byte
– char
– long
– short
– int
– float
– double
– reference
– returnAddress

Один «слот» зі змінною займає 4 байти. За винятком long і double, всі типи займають 1 слот в масиві локальних змінних. Дані типи займають 2 послідовних слова. JVM по своїй основі, як і Java є об’єктно-орієнтованою, тому в методах об’єкту 0 слот зарезервований під «this». Для статичних методів 0 слот є вільним для використання [2].

Окрім масиву, JVM також має окремий стек операндів, що за своїм функціоналом нагадує регістри процесора (дуже схожий на FPU стек x87 розширень). Більшість байт-коду розроблено для маніпуляцій із стеком оперантів, будь то: pop, dup, push, store, load. Тому досить часто доводиться переміщати дані між стеком та масивом локальних змін.

Загалом структура даних JVM є доволі простою, але помітною є деяка дуплікація даних з присутністю одночасно і масиву локальних змінних і стеку операндів.

1.1.1.2 Компоненти незалежні від потоків.

Hotspot використовує керуєму пам'ять, яку подібно до C називається Heap. Вона використовується для виділення пам’яті під масиви і класи під час виконання. Фрейми в JVM не підтримують динамічну зміну свого розміру після створення, тому об’єкти не можуть бути збереженими на стеку. На відмінну від примітивів об’єкти не знищуються коли метод закінчується. Об’єкти лише

виходять с пам’яті тоді, коди вони піддаються збиранню сміття [1]. Для підтримки даного механізму Heap ділиться на 3 секції:

а) «Нове покоління» (роділене на новачків (англ. Eden) на тих, що вижили (англ. Survivor));

б) «Старе покоління»;

в) «Постійне покоління».

JVM використовує алгоритм трасування для збирання сміття. Зазвичай це відбувається в наступному порядку [2]:

1. Нові об’єкти строюються в новому поколінні.
2. Однопоточне мале збирання прооперує лише нове покоління. Об’єкти, які вижили будуть переміщені в зону тих що вижили.
3. Велике збирання, що зупиняє роботу всіх покотів почне переміщати об’єкти між поколіннями. Об’єкти, що вижили будуть переміщенні із «нового» покоління в «старе», а із старого в «постійне».
4. При цьому старе покоління та постійне покоління збираються одночасно, коли одне із них заповниться.

Крім Heap пам’яті в JVM ще присутні об’єкти які є частиною роботи віртуальної машини. Ці об’єкти зберігаються в пам’яті, що називається Non-Heap memory. В ній зберігаються [2]:

– JIT кеш
– Зона методів
– Інтерновані рядки

Інтерпретація байт-коду не є настільки швидкою як виконання CPU інструкцій. Тому для підвищення швидкодії Hotspot VM збирає статистику про виконаний, та компілює найчастіше виконуваний код в нативний. Цей код зберігається в JIT кешу. JVM не виконує компіляцію абсолютно всього коду. VM використовує евристичні алгоритми для знаходження балансу між швидкістю виконання коду та часом необхідним для компіляції коду [2].

Зона методів на відмінну від її назви зберігає інформації про клас, таку як:

– Посилання на ClassLoader
– Посилання на пул констант
– Дані про поля класу
– Дані про методи класу
– Код методів та інформацію про їх виконання

Зона методів є спільною для всіх потоків. В більшості випадків вона використовується лише для читання, але специфікація JVM потребує потокобезпечного доступу до неї. Під час виконання її технічно можна змінити, а також специфікація не прописує [1], коли дана зона повинна бути ініціалізована. Тому реалізації JVM повинні забезпечити завантаження даної зони лише 1 раз, і лише всі інші потоки повинні чекати поки зона не буде завантажена.

Зона методів є структурою часу виконання. Вона ініціалізується із .class файлів, що є об'єктним представленням байт-коду. В які власне збирається Java та інші JVM мови. Структура .class файлу наведена на рисунку 1.2, а роз'яснення до неї в таблиці 1.2 [2].

```
ClassFile {
    u4                  magic;
    u2                  minor_version;
    u2                  major_version;
    u2                  constant_pool_count;
    cp_info             contant_pool[constant_pool_count – 1];
    u2                  access_flags;
    u2                  this_class;
    u2                  super_class;
    u2                  interfaces_count;
    u2                  interfaces[interfaces_count];
    u2                  fields_count;
    field_info fields[fields_count];
    u2                  methods_count;
    method_infomethods[methods_count];
    u2                  attributes_count;
    attribute_info      attributes[attributes_count];
}
```

Рис. 1.2 – Структура .class файлу

Таблиця 1.2 – Структура .class файлу

Назва поля	Опис
magic, minor_version, major_version	Інформація про версію класу на JDK для якої даний клас був скомпілюваний
constant_pool	Пул констант
access_flags	Список модифікаторів доступу для даного класу
this_class	Індекс в пулі констант що вказує на ім’я даного класу
super_class	Індекс в пулі констант що вказує на ім’я super класу
interfaces	Список з індексами в пулі констант інтерфейсів, що реалізовує даний клас
fields	Список з індексами в пулі констант, що вказують на повний опис всіх полів класу
methods	Список з індексами в пулі констант, що вказують на повний опис всіх методів класу та їх сигнатур. Якщо метод не абстрактний, то дане поле також зберігає його байт-код
attributes	Список з різними значеннями, що містять додаткову інформацію про клас, включаючи анотації з RetentionPolicy.CLASS або RetentionPolicy.RUNTIME

JVM перетворює файли .class в зону методів за допомогою спеціального об’єкту, що називається Classloader [1]. Зокрема, початкова ініціалізація перед виконанням public static void main(String[]) виконається за допомогою спеціального «Bootstrap Classloader», що реалізований в нативному коді [2]. Цей Classloader також значно швидший за інші так як він лише завантажує основні Java Api, які мають вищий ступінь довіри та не проходять велику частину валідації, що відбувається для користувацьких класів.

Крім Bootstrap Classloader JVM ще має System Classloader, що виконує безпосереднє завантаження класів користувача використовуючи CLASSPATH.

Крім цього, JVM надає можливість користувачу визначити всій Classloder із необхідною йому логікою.

Виконання даних об’єктів є досить дорогою операцією, одної із оптимізацій JVM – є зберігання класів, що часто використовуються в спеціальному архіві у пам’яті. Цей архів є одним на Операційну систему та може використовуватися усіма процесами JVM на комп’ютері [2].

Виконання main() уже приведе до завантаження, лінковки та ініціалізації інших класів та інтерфейсів за необхідністю. Схема підготовки .class файлу для виконання наведена на рисунку 1.3 [2].

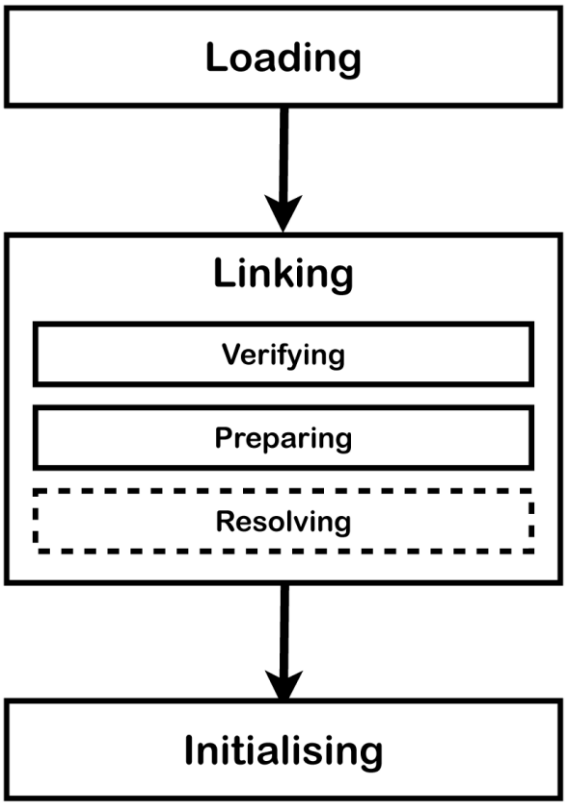


Рис 1.3 – Підготовка .class файлу до виконання

Лінковка в JVM складається із трьох частин перевірка, підготовка та необов’язкова, але реалізований в HotSpot: резолюція.

Перевірка – це процес перевірки того, що клас чи інтерфейс є структурно правильний і відповідає семантичним правилам мови Java та JVM в цілому.

Hotspot виконує на даному етапі наступні перевірки:

– Перевірка формату таблиць символів
– Перевірка того, що Final методи та класи не перевизначені
– Методи підкоряються правилам доступу Java
– Методи мають правильну кількість аргументів та типи
– Байткод виконує лише валідні операції зі стеком
– Ініціалізація змінних виконується перед їх читанням
– Змінні мають коректні значення для їх типів

Делегація перевірки до початку виконання означає, що під час виконання дані перевірки не є необхідними, але це збільшує час між запуском VM і виконанням першої інструкції [2].

Підготовка виконує ініціалізацію статичних структур даних самої JVM, таких як таблиці методів, а також статичних структур користувача в їхні значення за замовчуванням. Ініціалізатори статичних даних не запускаються. Вони виконуються як частина процесу ініціації трохи пізніше.

Резолюція це необов'язкова стадія яка перевіряє символічні посилання, що зберігаються в пулі констант на переводить їх в реальні посилання. Це забезпечує динамічне лінкування коду в JVM. Це відбувається шляхом завантаження всіх класів та інтерфейсів чиї ім'я буди знайдені в пулі констант. Специфікація JVM не потребує цієї стадії. Якщо вона відсутня то символічна резолюція відбувається при першому використанні посилання [2].

Ініціалізація проводиться шляхом виконання методу із спеціальною назвою <clinit>. Код в даному методі є першим байт-код що виконує JVM після запуску процесу.

JVM має 2 структури даних що зберігають інформації необхідні для виконання байт-коду, що не вміщаються в самі інструкції. Це уже пул констант та таблиця символів. Відмінність полягає в тому, що пул констант реалізований як масив та не змінюється під час виконання коду, а таблиця символів реалізована як хеш таблиця.

Пул констант є строго-типізованою колекцією та може зберігати лише обмежену кількість типів. Типи, що можуть зберігатися в пулі констант наведені в таблиці 1.3 [1].

Таблиця 1.3 – Типи даних пулу констант

Назва типу	Пояснення до типу
Integer	Int32 константа розміром в 4 байти
Long	Int64 константа розміром 8 байт
Float	Float32 константа розміром 4 байти
Double	Float64 константа розміром 8 байт
String	Посилання на Utf8 запис, що містить байти даного рядка.
Utf8	Набір байтів що містять послідовність символів Utf8
Class	Посилання на Utf8 запис, повне ім’я класу в специфічному JVM форматі
NameAndType	Розділена двокрапкою пара значень. Перше значення містить посилання на Utf8 запис з назвою методу чи поля. Друге значення містить посилання на Utf8 запис, що у разі коля – містить повну назву класу до якого належить дане поле, а у разі методу список повних назв параметрів даного методу.
Fieldref, Methodref, InterfaceMethodref	Розділена крапкою пара значень. Перше значення містить посилання на Class запис. Друге значення містить посилання на NameAndType

Таблиця символів представляє собою Хеш таблиця від посилання на символ до самого символу, тобто HashTable<&Symbol, Symbol>. Дана таблиця зберігає всі символи, що на даний момент використовуються JVM під час виконання, включаючи символи із пулу констант.

Приналежність символу до даної таблиці контролюється використовуючи механізм підрахунку посилань [2]. Наприклад, коли клас вивантажується, рахунок посилань усіх символів, що знаходяться в пулі констант даного класу,

зменшується на 1. Коли рахунок посилань символу в таблиці символів стає 0, то символ вивантажується із таблиці.

Крім цього специфікація мови Java вимагає, щоб однакові літерали типу String вказували на 1 і той же об’єкт String [1]. Задля цього літерали автоматично інтернуються компілятором. Крім того, користувач може викликати метод String#intern(), який поверне посилання на ново-інтернований рядок. У HotSpot інтернований рядок міститься у таблиці інтернації рядків, де ключом є посилання на об'єкт String, а значенням відповідний символ (тобто HashTable <&obj, Symbol>).

І для таблиці символів і для таблиці інтернації рядків, всі записи знаходяться в канонізованій формі Unicode, щоб забезпечити збереження лише 1 символу.

1.1.2 Структура байткоду JVM

Структура найбільш поширених інструкцій наведена на рисунку 1.4 [2].

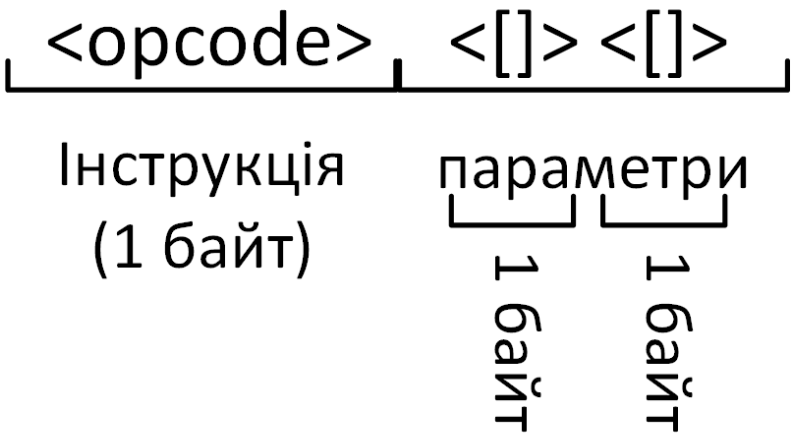


Рис. 1.4 – Структура інструкції JVM

Більшість інструкцій JVM не мають параметрів або мають лише 1 параметр. Інструкції з 2 параметрами це зазвичай зсув в байт-коді. Приклали інструкцій показані в таблиці 1.4 [3].

Таблиця 1.4 – Приклади інструкцій JVM

Назва інструкції	Код	Параметри	Пояснення
iadd	0x60	-	Цілочисельне додавання
aload	0x53	1: index	Завантаження посилання із масиву локальних змінних в стек
goto	0xa7	1: indexbyte 1, 2: indexbyte 2	Безумовний перехід

Крім того, JVM є так званий довгий режим (wide, інструкція c4), при якому деякі інструкції можна виконувати із параметрами розміром в 2 байти.

1.2 Віртуальна машина Lua 5.0

Lua 5.0 та стаття про її створення на сьогодні є класикою комп'ютерних наук. На відмінно від JVM віртуальна машина Lua має регістрову архітектуру. Стек досі існує – на ньому власне і живуть регістри, але у регістровій машині Lua можна посилатися не лише на вершину стеку, а й на будь-яку комірку, що існує в стеку [4]. Зазвичай регістрові машини асоціюють з більшим розміром байт-коду та складністю їх декодування. Але, нині це не є проблемою, а регістрова архітектура в цілому зменшує кількість інструкцій [4]. Крім того, регістрова машина буде працювати швидше, адже вона не тратить свій час на маніпуляції зі стеком, а декодування інструкцій особливо якщо їх параметри є цілими словами CPU не займає багато часу [4]. Також, на відмінну від Java в регістровій архітектурі не має необхідності в масиві локальних змінних – стек уже є масивом локальних змінним.

Для управління пам'яттю як і java використовується трасування, а точніше Lua використовує інкрементальне трасування, яке слідкує лише за певною кількістю об'єктів, але з кожним трасуванням, в залежності від заповненості пам'яті ця цифра збільшується.

1.2.1 Структура байткоду Lua 5.0

Байт-код Lua має досить цікаву структуру що значно відрізняється від структури байт-коду JVM та є спеціально оптимізованим під регістрову машину з сильно обмеженим списком інструкції. Одна команда Lua завжди займає 32 біти, але ці біти інтерпретуються по різному в залежності від бітів декілька режимів. В залежності від інструкції інші біти можуть бути один, двома, або трьома параметрами. Всі 4 режими інструкцій Lua VM показані на рисунках 1.5 – 1.8 [5].

Рис. 1.5 – Структура Lua інструкції в режимі iABC

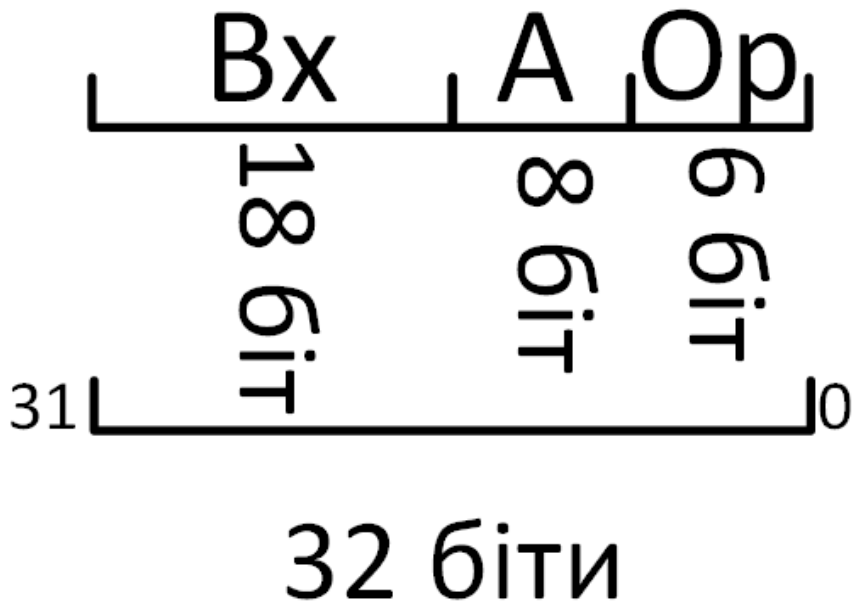


Рис. 1.6 – Структура Lua інструкції в режимі iABx

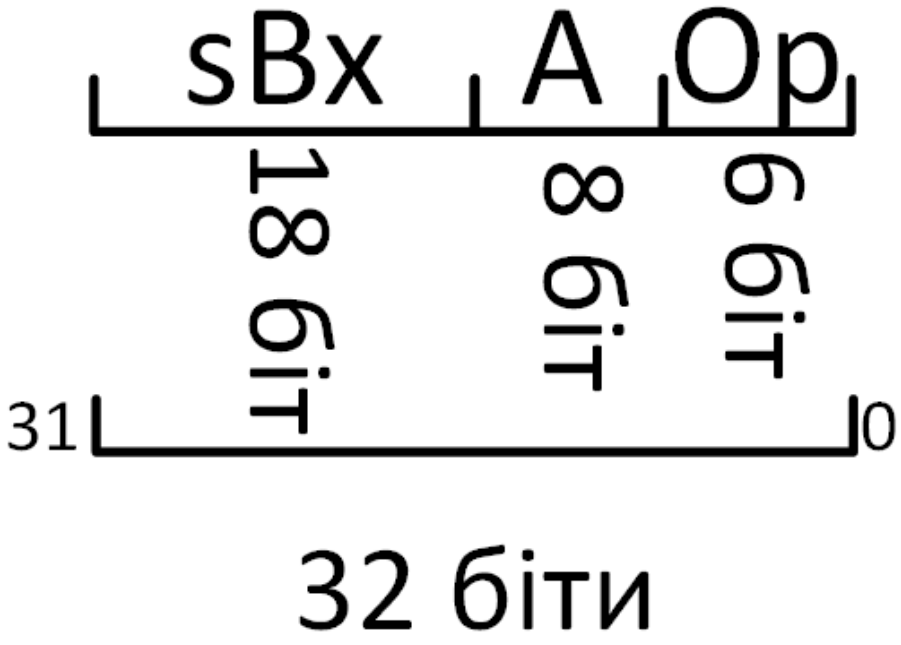


Рис. 1.7 – Структура Lua інструкції в режимі iAsBx

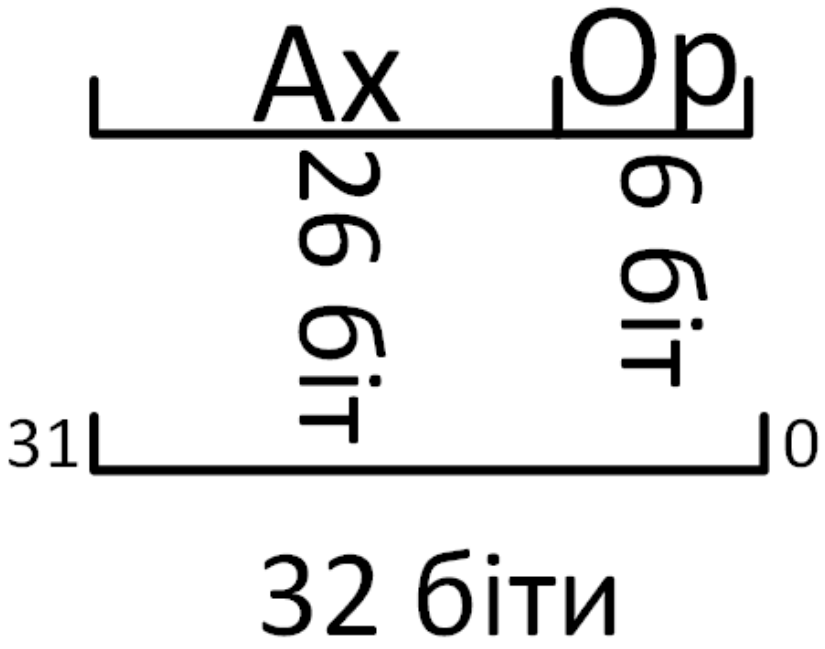


Рис. 1.8 – Структура Lua інструкції в режимі Iax

Відмінністю між sBx та Bx на рисунках 1.6, 1.7 є те, що sBx має знак, а Bx не має [4].

Варто також зауважити що інструкції в Lua кодуються в Litte Endian, тому 0 біт знаходиться зліва.

Віртуальна машина Lua 5.0 має лише 35 інструкцій. Всі інструкції, та їх короткий опис показані на рисунку 1.9 [4].

```
MOVE        A B      R(A) := R(B)
LOADK       A Bx     R(A) := K(Bx)
LOADBOOL    A B C    R(A) := (Bool)B; if (C) PC++
LOADNIL     A B      R(A) := ... := R(B) := nil
GETUPVAL    A B      R(A) := U[B]
GETGLOBAL   A Bx     R(A) := G[K(Bx)]
GETTABLE    A B C    R(A) := R(B)[RK(C)]
SETGLOBAL   A Bx     G[K(Bx)] := R(A)
SETUPVAL    A B      U[B] := R(A)
SETTABLE    A B C    R(A)[RK(B)] := RK(C)
NEWTABLE    A B C    R(A) := {} (size = B,C)
SELF        A B C    R(A+1) := R(B); R(A) := R(B)[RK(C)]
ADD         A B C    R(A) := RK(B) + RK(C)
SUB         A B C    R(A) := RK(B) - RK(C)
MUL         A B C    R(A) := RK(B) * RK(C)
DIV         A B C    R(A) := RK(B) / RK(C)
POW         A B C    R(A) := RK(B) ^ RK(C)
UNM         A B      R(A) := -R(B)
NOT         A B      R(A) := not R(B)
CONCAT      A B C    R(A) := R(B) .. ... .. R(C)
JMP         sBx      PC += sBx
EQ          A B C    if ((RK(B) == RK(C)) ~= A) then PC++
LT          A B C    if ((RK(B) <  RK(C)) ~= A) then PC++
LE          A B C    if ((RK(B) <= RK(C)) ~= A) then PC++
TEST        A B C    if (R(B) <=> C) then R(A) := R(B) else PC++
CALL        A B C    R(A), ... ,R(A+C-2) := R(A)(R(A+1), ... ,R(A+B-1))
TAILCALL    A B C    return R(A)(R(A+1), ... ,R(A+B-1))
RETURN      A B      return R(A), ... ,R(A+B-2)      (see note)
FORLOOP     A sBx    R(A)+=R(A+2); if R(A) <?= R(A+1) then PC+= sBx
TFORLOOP    A C      R(A+2), ... ,R(A+2+C) := R(A)(R(A+1), R(A+2));
TFORPREP    A sBx    if type(R(A)) == table then R(A+1):=R(A), R(A):=next;
SETLIST     A Bx     R(A)[Bx-Bx%FPF+i] := R(A+i), 1 <= i <= Bx%FPF+1
SETLISTO    A Bx
CLOSE       A        close stack variables up to R(A)
CLOSURE     A Bx     R(A) := closure(KPROTO[Bx], R(A), ... ,R(A+n))
```

Рис. 1.9 – Інструкції віртуальної машини Lua 5.0

На рисунку 1.9 присутні наступні умовні позначення:

– R(X) – регістр X

– K(X) – константа X із пулу констант

– RK(X) – якщо X < *k* то R(X), інакше K(X - *k*), де - *k* певна, залежна від реалізації константа, її значення за умовчанням 205.

– G[X] – значення X із глобальної таблиці Lua

– U[X] – верхнє значення X (параметр)

Регістри, як уже було сказано, зберігаються на стеку, що фактично є масивом, пул констант також є масивом. Верхні значення також зберігаються в масиві, що робить доступ до всіх ключових елементів віртуальної машини швидким.

ВИСНОВКИ ДО РОЗДІЛУ 1

Проаналізувавши існуючи системи виконання коду, зокрема JVM та Lua 5.0 Vm, були помічені недоліки та переваги обох систем. Зокрема обидві системи використовують трасування для управління пам’яттю. Крім вищезгаданого, ці системи розроблялися більш ніж 20 років та не враховують особливості сучасної техніки та можливості сучасних процесорів. Тому, було вирішено створити свою систему виконання коду, яка вбере в себе кращі ідеї існуючих систем, врахує особливості сучасних процесорів та буде використовувати новітню системи управління пам’яттю основану та циклі життя об’єкту.

РОЗДІЛ 2
ОГЛЯД АРХІТЕКТУРИ СИСТЕМИ

2.1 Особливості мови Rust

Реалізація систем виконання коду має сенс лише на мовах, що компілюються в CPU інструкції. Реалізація віртуальної машини, що запускаються в іншій віртуальній машині – це надто. Основними мовами на цьому ринку є C та C++. Але в них є проблеми. Сучасні методи розробки програмних продуктів показали їх некомпетентність та схильність до помилок при створення надійного програмного забезпечення. Але недавно вийшла нова мова програмування від Mozilla Foundation – Rust [6]. Rust займає цю ж саму нішу в мовах програмування. Але має більш сучасний синтаксис, систему типів та унікальний механізм забезпечення безпеки пам’яті, який власне і надихнув на створення даного дипломного проєкту. Розглянемо основні особливості мови, що допомагають у створенні даного проєкту.

2.1.1 Система типів Rust

Основними концепціями системи типів є Struct – структури. Enum – дискриміновані об’єднання та Trait – характеристики типів [7].

Структури поєднують певні значимі, логічно пов’язані данні. Приклад визначення структури показаний на рисунку 2.2 [7].

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
```

Рис. 2.1 – Приклад структури

На відмінно від структур в C, Rust не гарантує представлення структур в пам’яті [8]. Компілятор може розміщати члени структури так, як вважає за потрібне, або навіть розміщати невеликі структури повністю в регістрах процесора.

Якщо структури поєднують певні значимі данні, то дискриміновані об’єднання роз’єднують певні дані. Тобто, вони відображаються один із варіантів із певного набору допустимих значень. На відмінну від enum в C, в Rust enum може мати дані всередині. Гарним прикладом цього є enum IpAddr в стандартній бібліотеці. Скорчене визначення IpAddr зображене на рисунку 2.2 [8].

```
struct Ipv4Addr {
    // дані специфічні для V4
}

struct Ipv6Addr {
    // дані специфічні для V6
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

Рис. 2.2 – Визначення дискримінованого об’єднання IpAddr

Крім створення (збору) дискримінованого об’єднання, за допомогою match блоку, можна розібрати enum на його базові компоненти в залежності від його поточного стану. Для цього використовується конструкція match. Приклад розбору IpAddr наведений на рисунку 2.3.

```
match ip {
    IpAddr::V4(v4) => handle_v4(v4),
     IpAddr::V6(v6) => handle_v6(v6)
}
```

Рис. 2.2 – Розбір IpAddr

Розбір об’єднань є дуже потужним механізмом в мові Rust та дозволяє замінити велику кількість того, що довелося б реалізовувати поліморфізмом в Java, та великі витрати на сам поліморфізм простим відносним стрибком [7].

Структури та дискриміновані об’єднання можуть мати також необмежену кількість impl блоків. Функції визначені в impl блоці можуть бути викликані використовуючи синтаксис методів що звичний для програмістів C++, Java та інших об’єктно-орієнтованих мов. При цьому першим параметром повинно бути self, &self, &mut self та деякі інші спеціальні конструкції [8].

Базовий приклад impl блоків зображений на рисунку 2.3 [7].

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Рис. 2.3 – Приклад impl блоків для структур

Останньою та наймогутнішою частиною є характеристики типів – набір певного функціоналу, спільного між різними структурами на об'єднаннями. Так, наприклад, оператор «+» є нічим іншим, як характеристикою Add. Спрощене визначення характеристики Add наведене на рисунку 2.4, а приклад реалізації Add для власного типу зображене на рисунку 2.5 [8].

```
pub trait Add {
    #[must_use]
    fn add(self, rhs: Self) -> Self;
}
```

Рис. 2.4 – Trait Add

```
use std::ops::Add;

#[derive(Debug, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Self;

    fn add(self, other: Self) -> Self {
        Self {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

assert_eq!(Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
           Point { x: 3, y: 3 });
```

Рис. 2.5 – Реалізація trait Add для власного типу

Крім визначення спільного функціоналу для типів. Характеристики дозволяють розшити існуючи типи шляхом реалізації нової trait (як і impl блоки реалізації trait є незалежними від визначення самого типу). Характеристики також є невід'ємною частиною мономорфізму та поліморфізму в мові Rust [9].

2.1.2 Система модулів Rust

Rust має структуровану системи модулів які відображаються файлову систему. Приклад модулів показано на рисунку 2.6 [7].

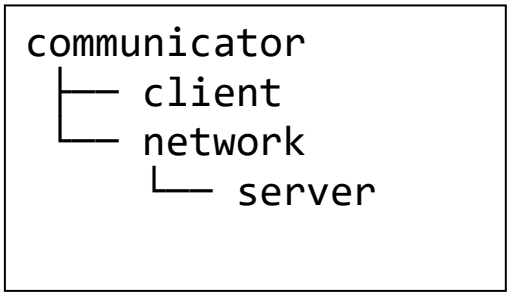


Рис. 2.6 – Приклад модулів Rust

Як можемо побачити з рисунку модулі утворюють ієрархічні структуру. Модулі описуються у файлах за допомогою ключового слова mod <назва модуля>. При цьому, якщо тіло модуля відсутнє то компілятор шукає модуль спочатку у папці с такою назвою, а потім у файлі. Кожен файл автоматично створює окремий модуль, а кожна папка з файлом «mod.rs» також є модулем. Приклад структури файлів, що відповідають модулям зображеним на рисунку 2.5 зображена на рисунку 2.7 [7].

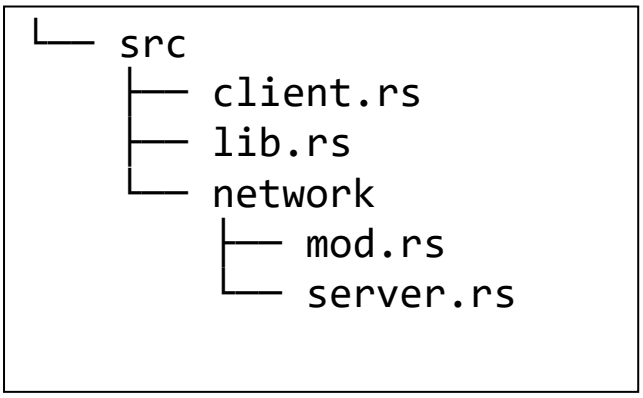


Рис. 2.7 – Структура фалів, що відповідає модулям

Rust дозволяє гранулярно обмежити область видимості членів модуля. За умовчанням члени модуля лише видні йому та його наслідникам. Використовуючи ключове слово pub можна зробити модуль видимим всім, а за допомогою pub(in <module>) обмежити область видимості до певного модуля в ієрархії.

Сама система модулів можливо не є дуже потужною, але за допомогою цієї системи під мовою програмування розрослася велика кількість бібліотек – ящиків (англ. Crate), які можна завантажити із crates.io та статично залінкувати в свою програму. Наразі crates.io налічує майже 40 тисяч ящиків [9]. Така можливість є небаченою для нативних мов програмування і значно прискорює та спрощує розробку складного функціоналу. Крім того, такий підхід дозволяє залишити стандартну бібліотеку Rust досить малою та винести функціонал, який не завжди є необхідним в окремі бібліотеки. Прикладами таких бібліотек є rand, libc, log, chrono та багато інших [9].

2.1.3 Абстракції з нульовою ціною

Основним принципом Rust як і C++ є принцип абстракцій з 0 ціною. Програміст не платить за те, що використовує. Крім очевидних речей, як наприклад стирання перечислень та структур в регістри. Основою цього принципу в Rust є характеристики [10]:

- Характеристика є єдиною суттю що представляє інтерфейс
- Характеристики можуть статично викликатися (мономорфізм)
- Характеристики можуть динамічно викликатися (поліморфізм)
- Характеристики дозволяють маркувати типи
- Характеристики дозволяють розширяти типи
- Характеристики є операторами

Останні 2 пункти уже були розглянуті. Розглянемо більш детально інші пункти.

Характеристики представляють інтерфейси типів. Подібно до Java та C# вони дозволяють абстрагуватися від певного типу. Але, на відмінно від цих мов, нові характеристики можуть бути визначені для існуючого типу. Тобто, абстракція над типом може бути створена після створення типу. На цьому все, характеристика є нічим іншим, чим абстракцією компілятора, під час виконання коду дана абстракція зникає.

Трохи цікавішою стає ситуація коли характеристики викликаються. Якщо викликати характеристику через дженерік, то абстракція зникає в процесі мономорфізму дженеріків і в результаті виконується статична резолюція функції. Приклад монорфізму характеристик наведений на рисунку 2.8 [10].

```
// функція
fn print_hash<T: Hash>(t: &T) {
    println!("The hash is {}",
t.hash())
}
// при виклику
print_hash(&true);
// буде скомпілявона в наступний код
__print_hash_bool(&true);
// після оптимізацій
println!(“{}”, 1);
```

Рис 2.8 – мономорфізм характеристик

Така модель має декілька переваг:

– Кожний виклик дженеріка дозволяє розмістити дані прямо в точці використання без додаткового непрямого виклику.
– Кожний виклик дженеріка дозволяє згенерувати спеціалізований код.

Як і C++ шаблони, дані переваги дозволяють згенерувати код специфічний під типи – написати краще вручну не вийде. Але, на відмінну від C++ дженеріки перевіряються під час написання, а не використання, що дозволяє отримати від компілятора кращі помилки [10].

Але, іноді абстракція є частиною представлення та не може бути стерта компілятором. В таку випадку програміст сам вибриє чи залишати йому абстракцію за допомогою ключового слова dyn. Приклад абстракції, що залишається наведений на рисунку 2.8 [10]:

```
fn print_hash(t: &dyn T) {
    println!("The hash is {}", t.hash())
    }
```

Рис 2.9 – поліморфізм характеристик

У випадку зображеному на рисунку 2.9 абстракція залишається в програмі та не зникає. Вибір правильної функції буде залежати від типу і це рішення буде прийняте під час виклику за допомогою механізму поліморфізму.

Таким чином, у Rust є лише 1 поняття інтерфейсу, що може бути використано для обох випадків із зарані відомою ціною.

Крім того, у Rust характеристики використовуються для маркування типів, ці маркери дають певну гарантію про типи. Так, наприклад маркер Send гарантує потокобезпечність типу. Маркери є дуже потужною властивістю Rust [8].

2.1.4 Безпека пам’яті

З поміж всього що було сказано Rust також гарантує безпеку пам’яті при використанні безпечного коду. Це забезпечують 3 системи: приналежність значення, позичання та тривалість життя [7].

Принцип досить простий: значення належить якийсь 1 адресі пам’яті. Якщо значення перемістити із цієї адреси в іншу, то значення по старій адресі вже невалідне. Даний механізм є ключовим для обох інших, а також є ключовим в підтримці Rust концепції: створення ресурсу і є ініціалізація ресурсу, що дозволяє обійтись мові без збирання сміття.

Так, як об’єкт може належати лише одній адресі то в Rust існують певні правила позичання об’єкту. Існує 2 типи посилань змінне (&mut) та незмінне (&). Привила позичання досить прості: в один момент в програмі може бути на об’єкт або 0 посилань, або 1 змінне посилання, або декілька незмінних посилань. При цьому, якщо існує хоч 1 посилання, то об’єкт не можна змінити (його можна змінити за &mut посиланням, якщо воно є). Це гарантує те, що об’єкт ніколи не буде в невалідному стані [7]. Всі покажчики, що належать об’єкту будуть вказувати на валідну адресу – невизначена поведінка не можлива.

Для посилань є ще одне правило: тривалості життя. У кожного об’єкта є тривалість життя. Посилання може існувати лише стільки часу скільки існує

сам об’єкт – посилання не може вийти за цю рамку. Тобто, наприклад, посилання на локальну змінну не можна повернути із функції. В більшості випадків компілятор може сам розібратися з тривалістю життя, але в складних ситуаціях, іноді, доводиться призначати посиланням тривалості життя самому програмісту за допомогою спеціального синтаксису «‘a», при цьому існує спеціальна тривалість життя – тривалість життя всієї програми, позначається «‘static». Даний механізм є ключовим в запобіганні «use after free» проблем [7].

Всі ці правила гарантують безпеку пам’яті. Але, іноді необхідно зробити щось, що виходить за рамки цих правил. Rust дозволяє вимкнути правила безпеки пам’яті за допомогою блоку «unsafe {} » [7]. Даний блок дозволяє використовувати покажчики, на які не розповсюджуються правила безпеки.

Зазвичай, програмісту не доводиться стикатися з даними концептами при написанні правильного та ідоматичного коду. Але, варто познайомитися з цими концептами, якщо працювати на границі можливостей системи типів і компілятора мови Rust [7].

2.1.5 Екосистема Rust

Крім потужного функціоналу самої мови Rust. Навколо мови була розроблена та підтримується потужна екосистема, що значно покращує розрубку в даній мові. Зокрема, частиною екосистеми є вищезгаданий crates.io. Важливою частиною екосистема є cargo система збирання коду, що стоїть поверх компілятора та дозволяє безболісно збирати ящики разом. Cargo також інкапсулює взаємодію із crates.io шляхом опису залежностей проекту в cargo.toml файлі. Приклад cargo.toml зображений на рисунку 2.7 [7].

```
[package]
name = "rusty_yard"
version = "0.2.2"
authors = ["hunter04d <hunter04d@gmail.com>"]
edition = "2018"

[profile.release]
lto = true

[dependencies]
lazy_static = "1.4"
thiserror = "1.0"

[dev-dependencies]
proptest = "0.9"
criterion = "0.3.0"
```

Рис. 2.10 – Cargo.toml для бібліотеки rusty_yard

Невід'ємною частиною cargo також є cargo doc – механізм зручної документації коду. Документація кожного ящика автоматично розміщується на docs.rs для зручного перегляду. Крім цього, детальна документація всього функціоналу мови та стандартної бібліотеки розміщена на doc.rust-lang.org, включаючи велику книгу Rust.

Rust також має добре розвинені механізми аналізу коду на основні помилки. Зокрема, це clippy – статичний аналізатор подібний clang-tidy та RLS – мовний сервер призначений для інтеграції з IDE.

2.2 Основні характеристики архітектури системи.

В ході реалізації даної системи будо вирішено зробити системо подібну до оглянутих в розділі 1 – віртуальну машину прикладного рівня зі своїм набором команд та власними обчислювальними можливостями.

Основним вибором для віртуальних систем виконання коду є вибір архітектури обчислювальних можливостей. Було розглянуто 2 можливих таких архітектури: стекова та регістрова. Із них була вибрана регістрова архітектура. Регістрова архітектура є більш подібною до архітектур сучасним процесорів, а також менше загальна кількість команд дозволяє компілятору краще оптимізувати кожну із команд що повинно пришвидшити швидкодію системи [15]. Стаття про створення Lua 5 також прийшла до таких же висновків [4].

Важливою характеристикою даної машини є її швидкий запуск. Потрібно щоб він запуску системи до виконання її першої інструкції пройшло як можна менше часу. Добитися цього 20 років назад і при цьому зберегти хорошу швидкодію систему будо неможливо.

Як в і в Java динамічне лінкування коду є основним механізмом зв’язування модулів системи разом за допомогою символічних посилань.

Оскільки ми не виконуємо валідацію байт-коду перед виконанням, то важливим є безпека кожної інструкції. Інструкція не новинна призвести до невалідного стану системи. Тому, кожна інструкція матиме ряд перевірок, що забезпечать коректне її виконання. У разі виконання невалідного байт-коду система повинна буде припинити виконання та повідомити користувача про проблему. 20 років назад таке архітектурне рішення дуже сильно б сповільнило систему. Але,гі на сьогодні кожний процесор оснащеним потужної технологією передбачення гілок, зокрема існує буде потужний та точний Tage Branch predictor від amd [16], але будь яких сучасний branch predictor повинен буде справитися із задачею без великих складнощів.

2.2.1 Структура байт-коду системи

Структура однієї команди байт-коду системи наведена на рисунку 2.11.

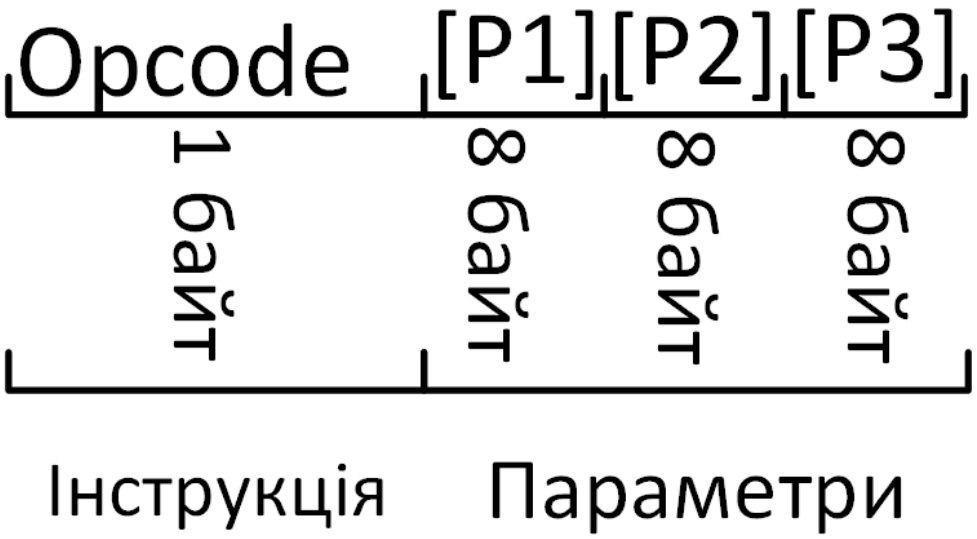


Рис. 2.11 – структура байт-коду системи

З рисунку видно, що пропонується структура інструкції при якій 1 байт займає сам ідентифікатор інструкції, і до 3 можливих параметрів розміром в 8 байт. Розміром параметрів було визначено 8 байт, адже це розмір 1 регістру x86-64 та Arm64. Таким чином, компілятор зможе виділити під кожний параметр окремий регістр. Звісно, така структура досить сильно збільшить розмір байт-коду, але з сучасними розмірами оперативної пам’яті – це не проблемою. Також, така схема значно спростить реалізацію.

Багато операцій є бінарними та міститимуть всі 3 параметри. Для унарних операцій необхідно лише 2 параметри, а інструкція безумовного переходу буде потребувати всього 1 параметр. Деякі службові операції будуть мати лише ідентифікатор операції та не міститимуть параметрів взагалі.

2.3 Механізм управління пам’яттю

2.3.1 Існуючі механізми управління пам’яттю

В основному існуючі системи виконання коду використовують один із 2 методів автоматичного управління пам’яті. Перший із них це трасування. Другий це підрахунок посилань.

Трасування являється найбільш поширеним метод трасування на реалізований в JVM та CLR.

Трасування полягає в проходження всього графу існуючих об’єктів, знаходженню об’єктів до який можна дійти від певних корінних місць. Зазвичай, в програмах – це стек та статичні дані.

Основною проблемою трасування є те, що називається «generational hypothesis» [11]. Більшість об’єктів в програмі мають дуже коротке життя. Тому в Java та .Net реалізовано поколінне збирання сміття. Нові об’єкти скануються на збираються досить часто, при цьому цей процес може відбуватися на іншому потоці, не припиняючи виконання програми. Але Generational GC є евристичним та може пропускати деякі об’єкти, тому навіть к такому GC час від часу потрібне повне трасування усього графу об’єктів системи [11].

Правильно реалізоване трасування може бути досить швидким. При виділені пам’яті, воно може навіть бути швидшим за алокацію на стеку. А збирання сміття може бути майже непомітним за винятком деяких повних трасувань час від часу. Але, основною проблемою трасування є його недетермінізм в часі виконання. Через так звані GC паузи – не можливо точно сказати, що коли програма завершить своє виконання, можна лише сказати, що це станеться колись.

Другим, більш простим та неефективним способом є підрахунок посилань. Суть цього методу полягає в тому, що кожен об’єкт має певне число – кількість активних посилань на нього в даний момент. Коли це число дійде до 0, то пам'ять цього об’єкту буде звільнена [12]. Даний спосіб являється дуже простим до реалізації, але при кожному новому посиланню на існуючий об’єкт необхідно збільшувати показник на 1, а при видаленню даного посилання зменшити. Хоча маніпуляції в одним числом не є дорогою операцією, але при звільнені великого графу об’єктів – ціна декременту дуже швидко додається [12].

Крім простоти реалізації, RC є досить потужною оптимізацією для функціональних мов. У функціональних мовах програміст маніпулює незмінними об’єктами, які часто повторюються. Середовище виконання може

мати таблицю інтернації об’єктів і таким чином кожен новий об’єкт, що повторює існуючий може указувати на старший об’єкт, при умові збільшення кількості посилань на нього [12].

Хоча RC має досить серйозні недоліки. Зокрема, наївна реалізація RC не знає як вийти із ситуації циклічних посилань[13]. Також, в багатопроцесорній системі рахунок посилань потребує атомарності. Виходом із першої проблеми є використання слабких посилань, або обмеженого трасування [12]. На жаль не існує рішення проблеми атомарності, тому багато віртуальних машин, що використовують підрахунок посилань мають певний глобальний монітор виконання, який не дозволяє більш як одному потоку виконуватися одночасно (наприклад Global interpreter lock в python).

Крім вищезгаданих проблем, проблемою збирання сміття є його недетермінізм стану пам’яті [11]. Не можливо знати коли точно ресурси будуть звільнені. Тому, в мовах Java, C#, Python – C++ концепція: створення ресурсу і є ініціалізація ресурсу (англ. Resource Acquisition is Initialization, скорочено RAII) замінена мовною конструкцією. Але, як результат, даного обмеження з’являється такий концепт як слабке посилання. Слабке посилання посилається на об’єкт, але через яке не відбувається трасування та воно не є частиною загального рахунку посилань [13]. Крім вищезгаданого, вирішення циклів в RC відбувається за допомогою слабких посилань. А також, даний концепт є ключовим у створенні слабких колекцій, які самі по собі не зберігаються об’єкти в пам’яті, а скоріше констатують факт їх існування.

Для вирішення основних проблем трасування та RC, зокрема їх недетермінізм та використані ресурси під час виконання, в даній роботі пропонується розгладнути інший спосіб управління пам’яттю оснований на життєвому циклі об’єкту.

2.3.2 Управління пам’яттю основане на циклі життя об’єктів

У об’єктів є певний життєвий цикл. Вони створюються зазвичай явним виділенням пам’яті програмістом, живуть та використовується певний час, а

потім певним чином знищуються. Єдиний спосіб гарантувати точний момент звільнення об’єкту – це ручне управління пам’яттю. Але, ми можемо загнати об’єкт в певні рамки – його «цикл життя». Цикл життя об’єкту – це певний обмежений блок коду в рамках якого об’єкт гарантовано буде існувати після його ініціалізації.

Цикли життя в програмі утворюють рекурсивну структуру, тобто один цикл може знаходитися в середині іншого. Найбільшим циклом життя – є цикл життя програми. Статичні дані маю цей цикл життя, оскільки вони існують протягом усього часу виконання програми. Усі об’єкти всередині певного циклу є валідними для цього циклу життя та для всіх циклів, що є нащадками. Коли цикл життя об’єктів закінчується, то всі об’єкти, що були створені у межах цього циклу знищуються у певному детермінованому порядку.

Для стекових систем виконання коду, цей цикл життя може позначатися спеціальними значенням на стеку. Це значення може з’явитися на стеку за допомогою спеціальної інструкції, а цикл може закінчитися окремою спеціальною інструкцією. Таким чином, після закінчення циклу, можна просто іти зверху в низ по стеку, до тих пір, поки не зустрінемо значення, що означало початок даного циклу. Цикл життя дуже добре лягає на існуючий механізм виклику функцій. Приклад закінчення циклу життя об’єктів в контексті повернення із функції наведений на рисунках 2.12 та 2.13.

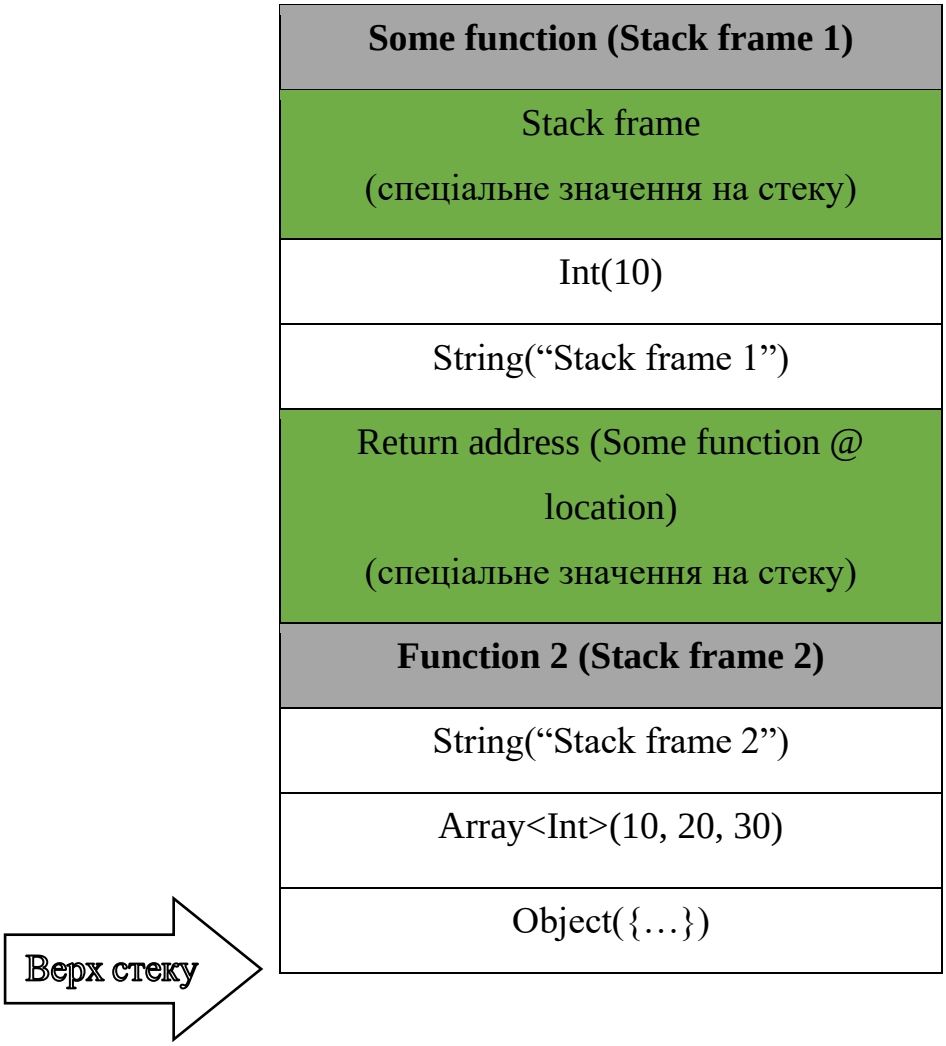


Рис. 2.12 – стек системи прямо перед виходом із функції.

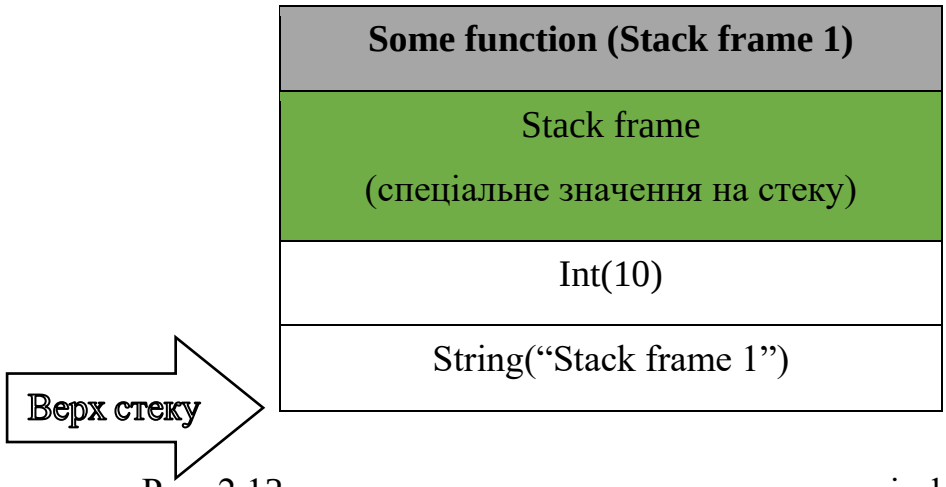


Рис. 2.13 – стек системи прямо перед виходом із функції.

На рисунку 2.12 бачимо, що перед виходом із функції, на стеку присутні усі локальні змінні функції. Під час виходу із функції, одночасно відбувається закінчення циклу життя цієї функції, тому ми ідемо по стеку поки не зустрінемо адресу повернення (вона в даному випадку служить спеціальним значенням на

стеку), звільнюючи стек та всі ресурси, що були задіяні локальними змінними функціями. Уже на рисунку 2.13 показаний стан системи після того як всі ресурси були звільнені.

Такий метод управління пам’яті гарантує існування певної точки у програмі після проходження якої певні ресурси будуть звільнені. Це досить близько до гарантування точного моменту звільнення пам’яті і значно краще, чим гарантії що дає нам трасування.

Але, у даної системи є серйозне обмеження. Задля забезпечення такої гарантії у кожного об’єкта повинен бути певний «корінь». Це певне місце в пам’яті, якому даний об’єкт приналежить і ніхто більше не має на нього посилання. Лише за допомогою цього кореня ми можемо гарантувати безпеку пам’яті такої системи за запобігти «use after free».

Тому, пропонується ввести механізм позичання подібний тому, що існує в Rust. Даний механізм полягає у наступних простих правилах:

а) Об’єкт може знаходитися в одному із трьох станів: без посилань (корінь є повним власником об’єкту), з одним змінним посиланням (може змінити об’єкт) та з декількома незмінними посиланнями, що не можуть змінити об’єкт

б) Поки на об’єкт є хоч одне посилання об’єкт не можна змінити за його коренем

в) Посилання на об’єкт повинно мати цикл життя, що строго менший за цикл життя об’єкту

г) Посилання можна створити лише в меншому циклі життя, чим сам об’єкт

Правила дуже подібні до тих, що в Rust. Останні 2 правила тісно пов’язані між собою. Передостаннє правило попереджає «use after free» (наприклад унеможливлює повернення посилання на локальне значення функції), а останнє правило призначене для того, щоб при закінченню циклу життя об’єкту корінь

був гарантовано єдиним власником об’єкту і не виникало проблем з порядком звільнення об’єктів.

Основною перевагою даного методу управління пам’яттю є детермінізм. Але він також дозволяє у віртуальних машинах обмежитися Heap областю програми і не строювати віртуальну пам'ять, на відмінну від методів трасування. Але, реалізація віртуальної пам’яті має сенс для зменшення фрагментації пам’яті. Також, рекомендується поміщати як змога більше даних на стеку задля зменшення фрагментації пам’яті. Реалізація в даній роботі не віртуалізує пам’ять, а використовує напряму пам’ять ОС, на якій запускається дана система, але все таки старається використовувати свій стек по максимуму.

Іншим недоліком є відсутність можливості створення рекурсивних структур через існування кореня об’єкту. В ситуаціях, коли це необхідно, можна надати програмісту засоби збирання сміття, щоб не обмежувати його в створені рекурсивних структур (RC зі слабкими посиланнями буде достатньо).

2.3.2.1 Процедура Deref

Останнім нерозкритим моментом в даному способі управління пам’яттю є те, як дістатися до значення, що зберігається за посиланням та зробити це безпечно. Для цього пропонується процедура Deref – призначена для тимчасового переміщення об’єкту для використання, а потім за необхідності повернення його назад.

Процедура має досить просту реалізацію: при виклику Deref на посиланні система іде за посиланням та копіює значення, що знаходиться за цим посилання у тимчасове місце. Система повинна буди уважною, щоб незмінні посилання не могли змінити це тимчасове значення, та якось вплинути на оригінальний об’єкт. Після закінчення використання значення за умови незмінного посилання: значення просто знищується, а при використанні змінного посилання копіюється назад – в те місце куди вказує посилання.

Дана процедура є безпечною, адже правила позичання гарантують, що ніхто крім змінного посилання не може модифікувати об’єкт, якщо таке

посилання присутнє. Для незмінних посилань дана процедура є безпечною лише при забезпечені того, що тимчасове значення не буде змінено, та ніяк не вплине на оригінальний об’єкт, тобто за умови гарантії повної незмінності оригінального об’єкту за цим посиланням (наприклад змінна значення, що зберігається за покажчиком в оригінальному об’єкті буде порушенням цього правила).

Варто зазначити, що дана процедура не є потокобезпечною для змінних посилань. При багатопоточній реалізації необхідно буде створювати для кожного об’єкту певний примітив синхронізації.

2.4 Система команд віртуальної машини

Важливою для розробки системи виконання коду є набір операцій, що можуть буди виконані та даній віртуальній машині. Далі пропонується розглянути запропоновану систему операцій із їх текстовим відображенням. Дана система команд включає як і операції необхідні для базової та ефективної обробки даних в системі, так і команди пов’язані з механізмом управління пам’яттю описаним в пункті 2.3.

Набір команд (інструкції) зображений в таблицях 2.1 – 2.12. Параметри нумеруються із 0, де 0 є першим після коду інструкції параметром та використовують спеціальні умовні позначення пояснення до умовних позначень наведені після команди.

Таблиця 2.1 – Інструкції завантаження

Команда	Параметри команди	Пояснення
U64Ld0		Завантаження в стек 0 типу U64
I64Ld0		Завантаження в стек 0 типу I64
LdTyped0	0: Type	Завантаження в стек 0 типу із параметру
LdType	0: Type 1: Value	Завантаження в стек значення із певним типом

Таблиця 2.1 (Закінчення)

Команда	Параметри команди	Пояснення
LdUnit		Завантаження в стек Unit типу
LdTrue		Завантаження в стек true типу bool
LdFalse		Завантаження в стек false типу bool
LdSS	0: Value	Завантаження в стек посилання на статичний рядок

Дані інструкції із таблиці 2.1 завантажують значення із пулу констант на верх стеку. Умовним позначенням параметрів Value – це значення, що завантажується а Type це тип даного значення.

Таблиця 2.2 – Інструкції без знакових операції

Команда	Параметри команди	Пояснення
UAdd	0: res 1: op1 2: op2	Цілочисельне додавання без знаку
USub	0: res 1: op1 2: op2	Цілочисельне віднімання без знаку
UMul	0: res 1: op1 2: op2	Цілочисельне множення без знаку
UDiv	0: res 1: op1 2: op2	Цілочисельне ділення без знаку
URem	0: res 1: op1 2: op2	Цілочисельна остача від ділення без знаку

Таблиця 2.3 – Інструкції знакових операції

Команда	Параметри команди	Пояснення
IAdd	0: res 1: op1 2: op2	Цілочисельне додавання без знаку
ISub	0: res 1: op1 2: op2	Цілочисельне віднімання, зі знаком
IMul	0: res 1: op1 2: op2	Цілочисельне множення, зі знаком
IDiv	0: res 1: op1 2: op2	Цілочисельне ділення, зі знаком без знаку
IRem	0: res 1: op1 2: op2	Цілочисельна остача від ділення, зі знаком
INeg	0: res 1: op	Протилежне за знаком число
IMod	0: res 1: op	Модуль числа

В таблицях 2.2 та 2.3 наведені операції для цілочисельних типів із знаком та без нього. Умовним позначенням параметрів тут є res – регістр результату операції, та op* - регістри операндів. Хоча на x86-64 та ARM64 знакові та беззнакові операції виконуються однаково, причиною розділу інструкцій на знакові та без знакові – є зменшення помилок програміста та не неприв’язана до існуючих представлень даних архітектура системи.

Таблиця 2.4 – Інструкції операцій із плаваючою точкою

Команда	Параметри команди	Пояснення
FAdd	0: res 1: op1 2: op2	Додавання
FSub	0: res 1: op1 2: op2	Віднімання
FMul	0: res 1: op1 2: op2	Множення
FDiv	0: res 1: op1 2: op2	Ділення
FRem	0: res 1: op1 2: op2	Остача від ділення
FNeg	0: res 1: op	Протилежне за знаком число
FMod	0: res 1: op	Модуль числа

В таблиці 2.4 наведені операції для типів із плаваючою точкою. Реалізація даних типів та операцій слідує стандарту IEEE 754 [14]. Умовні позначення тут такі ж, як і в минулих командах.

Таблиця 2.5 – Інструкції булевих операцій

Команда	Параметри команди	Пояснення
BAnd	0: res 1: op1 2: op2	Булеве І
Bor	0: res 1: op1 2: op2	Булеве Або

Таблиця 2.5 (Закінчення)

Команда	Параметри команди	Пояснення
Bnot	0: res 1: op	Булеве НІ
Bbe	0: res 1: op	Булева функція відображення

В таблиці 2.4 наведені булеві операції. В даній віртуальній машині булеві операції подібні до тих, що присутні в мові С та можуть буди застосовані для всіх чисельних типів. Наприклад BBe <значення>10 поверне тип bool зі значенням true. Умовні позначення тут такі ж, як і в минулих командах.

Таблиця 2.6 – Інструкції логічних (бітових) операцій

Команда	Параметри команди	Пояснення
Land	0: res 1: op1 2: op2	Логічне І
Lor	0: res 1: op1 2: op2	Логічне Або
Lnot	0: res 1: op	Логічне Ні
Lxor	0: res 1: op1 2: op2	Логічне виключне Або

Таблиця 2.7 – Інструкції логічних операцій здвигів та поворотів

Команда	Параметри команди	Пояснення
Shl	0: res 1: op1 2: op2	Здвиг вліво
Shr	0: res 1: op1 2: op2	Здвиг вправо

Таблиця 2.7 (Закінчення)

Команда	Параметри команди	Пояснення
RotL	0: res 1: op1 2: op2	Поворот вліво
RotR	0: res 1: op1 2: op2	Поворот вправо

В таблицях 2.6 та 2.7 наведені логічні операції. В даній віртуальній машині логічні операції подібні до тих, що присутні в мові С. Команди Shl та Shr виконують арифметичний здвиг для типів зі знаком та логічних для типів без нього. Крім них, також, є реалізованими операції RotL – бітовий повороти вліво та RotR – бітовий поворот вправо. Хоча дані команди можна реалізувати через здвиги, присутність даних команд може допомоги в розробці та не розкриває деталей реалізації типів в середині віртуальної машини. Умовні позначення тут такі ж, як і в минулих командах.

Таблиця 2.8 – Інструкції операцій порівнянь

Команда	Параметри команди	Пояснення
Ge	0: res 1: op1 2: op2	Оператор «>=»
Gt	0: res 1: op1 2: op2	Оператор «>»
Le	0: res 1: op1 2: op2	Оператор «<=»
Lt	0: res 1: op1 2: op2	Оператор «<»

Таблиця 2.8 (Закінчення)

Команда	Параметри команди	Пояснення
Eq	0: res 1: op1 2: op2	Оператор «==»
Ne	0: res 1: op1 2: op2	Оператор «!=»

В таблиці 2.8 показані команди порівнянь. Було прийнято рішення роз’єднати порівняння значень від стрибків. Це дозволить виконати більшість сценаріїв стрибків у лише дві команди, а також дозволить використати порівняння поза стрибками що значно спростить їх використання у порівнянні з байт-кодом Java. Умовні позначення тут такі ж, як і в минулих командах.

Таблиця 2.9 – Інструкції переходів

Команда	Параметри команди	Пояснення
J	0: index	Безумовний перехід
JC	0: index 1: value	умовний перехід

В таблиці 2.8 показані команди умовного переходу. Завдяки роз’єднанню переходів від команд порівнянь для даної системи достатньо лише 2 інструкції переходу. Команда J виконує безумовний перехід за здвигом від початку байткоду вказаного в параметрі, а команда JC виконує умовний перехід. Перший параметр – такий же як і в J. Другий параметр – вказує на регістр із булевим значенням. Перехід буде виконаний лише тоді, коли значенням регістру є true типу bool.

Таблиця 2.10 – Інструкції роботи із пам’яттю

Команда	Параметри команди	Пояснення
StartScope		Початок нового блоку циклу життя об’єктів
EndScope		Кінець блоку циклу життя об’єктів
TakeRef	0: Value	Взяття незмінного посилання
TakeMut	0: Value	Взяття змінного посилання
StartDeref	0: Value	Початок процедури Deref посилання
EndDeref		Закінчення процедури Deref

В таблиці 2.9 наведені операції роботи із пам’яттю. Створення нового циклу життя, а також процедури Deref. Варто помітити, що Deref потребує двох команд: одну для початку і одну для закінчення. Інструкція StartDeref завантажує нове тимчасове значення на стек. Дане значення не можна перемістити та на нього не можна взяти посилання. EndDeref знищує це тимчасове значення та поміщає повертає його назад (за необхідності). Умовним позначенням Value тут зображується параметр інструкції – регістр з яким відбувається операція.

Таблиця 2.11 – Інструкції роботи з функціями

Команда	Параметри команди	Пояснення
Call	0: Module 1: Fn	Виклик функції
Ret	0: Value	Повернення із функції

В таблиці 2.11 описані команди роботи із функціями. Call викликає функцію із модуля Module із назвою Fn. Ret повертає значення із функції, що знаходиться в регістрі Value. Команда call також проводить дії зі стрворення нового блоку, побідні до StartScope, а Ret виконує дії подібні до EndScope.

Таблиця 2.12 – Інструкції роботи з масивами

Команда	Параметри команди	Пояснення
SArrCreate0	0: Size 1: Type	Створення статичного масиву ініціалізованого 0 значеннями
SArrRef	0: Array 1: Index	Взяття значення із статичного масиву за незмінним посиланням
SArrMut	0: Array 1: Index	Взяття значення із статичного масиву за змінним посиланням
SArrSet	0: Array 1: Index 2: Value	Встановлення значення в масиві
SArrXcg	0: Array 1: Index 2: Value	Заміна значення с масиві на нове із регістру, та переміщення існуючого значення в регістр

В таблиці 2.12 показані команди роботи з масивами. Масиви є статичними структурами даних та зберігаються повністю на стеку. Варто зауважити, що значення не можна забрати назад після його запису в масив через SArrSet. Хоча замість SArrSet можна використати SArrXcg в разі потреби у старому значенні. SArrGet та SArrMut додає у стек посилання на значення за індексом та замикає весь масив відповідним замком. Відповідно, дані операції можуть бути виконані лише в новому блоці після масиву. Значення будь-якого елемента масиву не можна змінити, якщо замок є встановленим на масив. Умовними позначеннями тут є:

- Array: регістр в якому зберігається масив
- Size: розмір масиву, закодований прямо в інструкції
- Index: індекс в масиві починаючи з 0. закодований прямо в інструкції
- Value: регістр із якого береться значення для SArrSet

Таблиця 2.13 – Інструкції що маніпулюють стек

Команда	Параметри команди	Пояснення
Pop		Зняття останнього значення із стеку
Mv	0: Value 1: NewValue	Переміщення значення із одного регістра в інший, типи мають співпадати
Remove	0: Value	Дана команда знищить значення у стеку за індексом Value, всі інші значення будуть переміщенні щоб заповнити старе значення, можна застосувати лише для поточного блоку

В таблиці 2.13 наведені операції зі стеком. Необхідно зауважити, що Remove має лінійну складність та потребує виділення пам’яті, у зв’язку із семантикою позичань в мові Rust. Умовним позначаннями тут є Value – старий регістр, або регістр з яким відбувається операція. NewValue – регістр в якому зберігається нове значення регістр.

ВИСНОВКИ ДО РОЗДІЛУ 2

В даному розділі були розглянуті основні архітектурні особливості системи. Зокрема, була розглянута мова програмування Rust, описані її основні особливості, які роблять її ідеальної для написання системи виконання коду, такі як: абстракції з 0 ціною, екосистема та безпека пам’яті.

Також, були розглянуті основні можливі варіанти виконання архітектури системи. Із розглянутих: була вибрана регістрова архітектура виконання команд та JVM подібна структура байт-коду.

Був представлений метод та набір правил для управління пам’яттю, шляхом управління циклом життя об’єкту. Основною перевагою даного методу є певний детермінізм, але разом з цим є ряд недоліків, основним з який є ускладнення життя програмісту у певних ситуаціях.

З урахуванням архітектури системи, та враховуючи новий принцип управління пам’яттю, був розроблений спеціальний набір інструкції системи, який задовольняє архітектурні особливості як самої системи, так і сучасних процесорів, для яких в основному розробляється дана система.

РОЗДІЛ 3
ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Структура модулів програми

Система реалізована як бібліотека Rust.

Ієрархічна структура основних модулів реалізації зображена на рисунку 3.1.

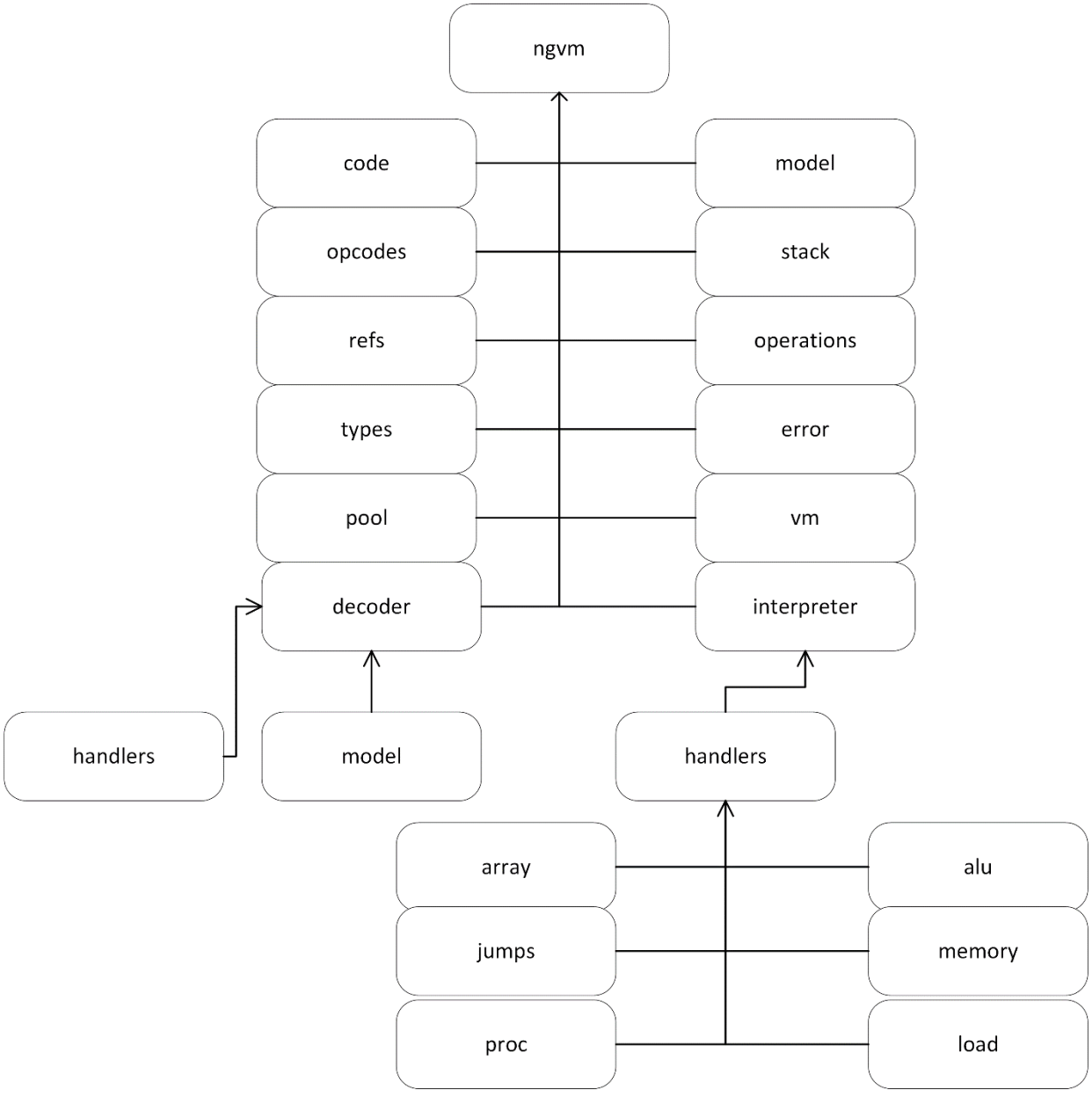


Рис. 3.1 – структура основних модулів

Завдяки реалізації системи як бібліотеку, це дозволяє вставляти систему в існуючі проєкти, наприклад в рушії ігор, або існуючі вбудовані рішення, а також додавати певні функції самому, за допомогою потужної системи типів Rust.

3.2 Ключові особливості модулів програми

Розглянемо основні модулі програми та їх ключові особливості більш детально.

Модуль *code* – представляє собою структуру та набір основних операцій із декодування байт-коду. Сам тип Code не є цікавим – це лище newtype над байтами, ключовим у цьому модулі є тип Chunk завдяки якому, декодування кожної нової інструкції відбувається так, як будь-то це перша інструкція у програмі. Опис даного типу наведений на рисунку 3.2.

```
pub struct Chunk<'a> {
    pub(super) bytes: &'a [u8],
    pub(super) offset: usize,
}
```

Рисунок 3.2 – опис типу Chunk

Як бачимо з рисунку 3.2 цей тип має посилання на байти – власне сам код системи та зсув у цьому коді. Саме завдяки зсуву, кожна нова інструкція читається, як перша інструкція.

Модуль *opcodes* містить у собі високорівневе представлення байт-коду системи як перечислення. Дане перечислення містить всі інструкції описані в розділі 2, а також ідентифікатори цих інструкцій в байт-коді.

Модуль *model* – представляє собою висорівневу об’єктну модель байт-коду системи, та є єдиним джерелом правди для байт-коду. Приклад байт-коду описаного на даній об’єктній моделі представлено на рисунку 3.3.

```
vec![
      LDType {
          type_location: PoolRef(0),
          value_location: PoolRef(1),
          },
       LDType {
          type_location: PoolRef(0),
           value_location: PoolRef(2),
       },
       LdTyped0 { type_location: PoolRef(0) },
       FAdd(three(2, 0, 1)),
       TraceStackValue(one(2)),
]
```

Рисунок 3.3 – опис байткоду на об’єктній моделі

Модуль *refs* абстрагує над параметрами інструкції та подає їх за допомогою уніфікованої моделі типів посилань. Основні типи цього модуля наведені на рисунку 3.4

```rust
pub type Ref = usize;

pub struct StackRef(pub Ref);

#[derive(Debug, Eq, PartialEq, Copy, Clone)]
pub struct PoolRef(pub Ref);

#[derive(Debug, Eq, PartialEq, Copy, Clone)]
pub enum CodeRef {
    Stack(StackRef),
    Pool(PoolRef),
    Offset(usize),
}
```

Рисунок 3.4 – Основні типу модуля refs

Як видно з рисунку 3.4 , крім слаботипізованого типу Ref цей модуль також має строготипізовані посилання на Стек та на пул констант, а також тип вбудованого значення під назвою Offset. Також даний модуль має об’єднання

даних посилань в типі CodeRef. Дана архітектура допомогла зменшити кількість помилок під час розробки даної системи.

Модуль *types* – це модуль в якому описані основні типи, що присутні в даній системі. Існує 2 види типів: примітиви та типи покажчики. Список типів примітивів та їх аналогів в мові Rust наведено в таблиці 3.1.

Таблиця 3.1 – Примітиви системи

Назва типу	Розмір (в комірках стеку)	Еквівалент в Rust
U64	1	u64
U32	1	u32
U16	1	u16
U8	1	u8
I64	1	i64
I32	1	i32
I16	1	i16
I8	1	i8
F32	1	f32
F64	1	f64
Bool	1	bool
Unit	1	()
Never	1	!
SStr	2	&’pool str
RetAddr	3	внутрішній тип системи

Крім примітивів в системі також наявні вказівні типи. Дані типи не є типами самі по собі, але можуть стати такими якщо заповнити їх іншими типами. Це щось подібне до дженериків в Java, але більш строго типізоване. Дана абстракція була розроблена спеціально для підтримки посилань, але

підійшла також для реалізації масивів. Список реалізованих вказівних типів наведено в таблиці 3.2.

Таблиця 3.2 – Вказівні типи системи

Назва типу	Розмір (в комірках стеку)	Еквівалент в Rust
SArr	1 + k * n k - розмір масиву n – розмір типу покажчика	[T; n]
Ref::Ref	1	&’a T
Ref::Mut	1	&’a mut T

Модуль *stack* представляє собою 2 типи. Слаботипізований StackData – комірка стеку, та строго типізований тип StackMeta що містить інформацію про комірку стеку. Опис типу StackData та скорочений опис типу StackMeta неведений на рисунку 3.5.

```
pub type StackData = [u8; 8];

#[derive(Debug)]
pub struct StackMeta {
    pub value_type: VmType,
    pub index: StackRef,
    pub metaflags: Metaflags,
}
```

Рис 3.5 – Основні типи модулю stack

Модуль *error* містить у собі дискриміноване об’єднання усіх можливих помилок системи та є єдиним джерелом помилок системи. Зручний інтерфейс даного типу був реалізований за допомогою бібліотеки this_error.

Модуль *operations* представляє собою уніфіковані абстракції роботи над примітивами в мові Rust. Хоча даний модуль не є частиною реалізації системи в цілому, але був ключим для реалізації арифметично-логічного блоку

інтерпретатора. Даний модуль складається із 4 характеристик. Вони представлені на рисунку 3.6.

```rust
pub trait BiOpMarker: Debug {}

pub trait BiOp<M: BiOpMarker, Other = Self> {
    type Output;

    fn invoke(self, other: Other) -> Self::Output;
}

pub trait UOpMarker {}

pub trait UOp<M: UOpMarker> {
    type Output;

    fn invoke(self) -> Self::Output;
}
```

Рис 3.6 – характеристики модулю operations

Як бачимо в даному модулі описані характеристика операції та характеристика маркеру операції. Це реалізовано шляхом створення Unit структур – маркерів та покритих реалізацій характеристик операцій від типів операцій, що є характеристиками стандартної бібліотеки мови Rust. Завдяки цьому вдалося створити уніфікований спосіб обробки операцій над примітивами.

Модуль *pool* містить у собі дискримінованне об'єднання – строго типізовану колекцію що представляє собою пул констант системи. Пул констант системи містить наступні типи приведені в таблиці 3.3

Таблиця 3.3 – Типи в пулі констант

Тип	Пояснення
Value	Нетипізоване значееня буль-якого типи розміром до 16 байт
String	Utf -8 послідовність символів
Type	Перний тип віртуальної машини (не може буди вказівним)

Таблиця 3.3 (Закінчення)

Тип	Пояснення
ModRef	Символічне посилання на модуль. Містить повну зазву модуля та його версію для динамічного лінкування
FnRef	Символічне посилання на функції. Містить повну назву функції для динамічного лінкування.

Ключовим в модулі *Vm* є власне вам тип Vm. Від представляє собою саму віртуально машину. Хоча тип Vm досить громісткий, даний модуль містить у собі методи Vm призначенні для швидкої ініціалізації Vm та початку виконання. Ключовою особливістю даної віртуальної машини є її незалежність від контексту виконання від коду що виконується. Таким чином можна виконувати код, що не є валідною частиною модулів завантажених у віртуальну машину. Ця можливість з'явилася завдяки безпечній архітектурі даної системи. Скорочений опис структури VM наведений на рисунку 3.7.

```
pub struct Vm {
    /// vm stack values
    pub(crate) stack: Vec<StackData>,
    /// vm stack metadata
    pub(crate) stack_metadata: Vec<StackMetadata>,
    /// current instruction in the current stack frame
    pub(crate) ip: usize,

    /// Index from which all indexing is happening
    pub(crate) last_stack_frame: usize,

    /// Index of the last pushed value
    pub(crate) last_pushed_value: usize,
    /// Loaded modules
    pub(crate) modules: HashMap<ModRef, Module>,

    pub(crate) current_module: ModRef,

    pub(crate) current_fn: FnRef,

    pub(crate) cycle: usize,
}
```

Рис 3.7 – структура Vm

Модулі *decoder* та *interpreter* дуже схожі за своєю структурою. У них обох є під-модуль *handler* який взаємодіє із байт-кодом. Різниця між ними полягає у тому, що *decoder* – транслює байт-код в структуру призначену для текстового відображення байт-коду (опис об'єктної моделі якої знаходиться в під-модулі model), а *interpreter* інтерпретує байт-код з використанням структури Vm. Основним в обох модулях є масив *handler*. В даному масиві зберігаються функції обробники байт-коду, причому за індексом *i* в даному масиві зберігається обробник інструкції з ідентифікатором *i*. Така структура дозволяє виконувати відповідні операції шляхом лише 1 непрямого стрибка.

Під-модуль *handler* в *interpreter* також розбитий на під-модулі, що містять в собі певний перелік обробників інструкцій. Опис цих під-модулів, а також типів інструкцій що вони обробляють наведено в таблиці 3.4.

Таблиця 3.4 – Підмодулі interpreter::handler

Підмодуль	Опис
alu	Містить обробники арифметично-логічних операції. Дані обробники реалізовані за допомогою абстракцій з модуля operations
array	Обробляє операції пов'язані з масивами
jumps	Обробляє умовні та безумовні переходи
memory	Обробляє команди що пов'язані з управлінням пам'яті в системи на основі методу управління життєвим циклом об'єктів
proc	Обробляє команди по входу у функції та повернення з них
load	Обробляє команди завантаження даних з пулу констант

Як видно з таблиці 3.4, модулі розбиті за логічно пов'язаними інструкціями. Так як всистема використовує регістрову модель, то в деяких випадках обробка однієї інструкції потребує досить великої кількості дій, тому було прийнято рішення про розбиття interpreter::handler на декілька під-модулів.

ВИСНОВКИ ДО РОЗДІЛУ 3

У даному розділу представлена модульна структура реалізації системи та ключові особливості кожного із модулів. Було показано, що мова Rust є ідеальною для реалізації таких систем для програміста завдяки глибоким можливостям системи типів мови.

Крім цього, були продемонстровані деякі дизайнерські рішення, що були прийняті під час реалізації даної системи. Такими рішеннями є: вказівні типи - завдяки ним реалізовані посилання, та об’єктна модель байт-коду – вона стала основним способом представлення байт-коду, замість звичайних байт.

Побудована система реалізує інструкції що були описані в розділі 2, а також показує, що реалізація методу управління пам’яті є реальною на практиці.

РОЗДІЛ 4
АНАЛІЗ ЕКСПЕРЕМЕНТАЛЬНОЇ РЕАЛІЗАЦІЇ

4.1 Виклик системи

Система розроблена в основному для вбудованих сценаріїв, тому основний спосіб виклику системи це – статично залінкований crate за допомогою системи збору проєктів cargo. Для цього необхідно додати наступний рядок до Cargo.toml:

```
[dependencies]
rand = { git = "https://github.com/hunter04d/ngvm" }
```

Рис 4.1 – Додавання системи до існуючого проєкту

Після даних дій описаних на рис 4.1 система буде статично залікована до необхідного проєкту.

4.1.1 Способи задання коду

Як було описано в розділі 3 – байт-код не є основним способом задання коду для системи. Система використовує спеціальну об’єктну модель для якої існують наступні способи задання:

– Програмно як Vec<Opcode>
– За допомогою yaml файлу
– За допомогою json файлу
– За допомогою стиснутого bincode формату

Програмне задання коду – це задання коду прямо в програмі на мові Rust. Такий спосіб є дуже зручним, адже підтримує базову перевірку правильності написаного коду.

Так код, що виводить на екран перші n чисел Фібоначчі можна задати за допомогою наступної функції в Rust.

```
vec![
        Ld0U64,
        Ld0U64,
        LDType {
            type_location: PoolRef(0),
            value_location: PoolRef(2),
        },
        LDType {
            type_location: PoolRef(0),
            value_location: PoolRef(2),
        },
        LDType {
            type_location: PoolRef(0),
            value_location: PoolRef(1),
        },
        LDType {
            type_location: PoolRef(0),
            value_location: PoolRef(2),
        },
        LdFalse,
        Ld0U64,
        Label(0),
        TraceStackValue(s(2)),
        UAdd(three(0, 1, 2)),
        Mv(s(1), s(2)),
        Mv(s(2), s(0)),
        UAdd(three(3, 3, 5)),
        Le(three(6, 3, 4)),
        JC {
            label: 0,
            cond: s(6),
        },
    ]
```

Рис 4.2 – Код представлений як функція в Rust

Замість розробки специфічного та складного до розуміння текстового формату для коду, система дозволяє задавати код в вигляді yaml файлу. Такий формат досить близько слідує програмному формату та є зручним для розуміння, читання та написання. Приклад цього ж самого коду у вигляді yaml файлу наведено на рисунку 4.3.

```yaml
---
- Ld0U64
- Ld0U64
- LDType:
    type_location: 0
    value_location: 2
- LDType:
    type_location: 0
    value_location: 2
- LDType:
    type_location: 0
    value_location: 1
- LDType:
    type_location: 0
    value_location: 2
- LdFalse
- Ld0U64
- Label: 0
- TraceStackValue: 2
- UAdd:
    result: 0
    op1: 1
    op2: 2
- Mv:
    - 1
    - 2
- Mv:
    - 2
    - 0
- UAdd:
    result: 3
    op1: 3
    op2: 5
- Le:
    result: 6
    op1: 3
    op2: 4
- JC:
    label: 0
    cond: 6
```

Рис 4.3 – Код представлений як yaml файл

Крім 2 людських форматів, код можна представити за допомогою 2 машинних форматів.

Json є дуже зручним та простим для обробки у фактично будь-яких умовах, тому у умовах зберігання коду на сторонніх пристроях система надає можливості виконувати код із Json файлу. Хоча Json вважається машинно-людським форматом, представлення коду в Json розроблено для простоти машинного розуміння, та не є зручним для людини. Той же самий код що і на 2 минулих рисунках наведено в Json форматі на рисунку 4.4.


```
["Ld0U64","Ld0U64",{"LDType":{"type_location":0,"
value_location":2}},{"LDType":{"type_location":0,
"value_location":2}},{"LDType":{"type_location":0
,"value_location":1}},{"LDType":{"type_location":
0,"value_location":2}},"LdFalse","Ld0U64",{"Label
":0},{"TraceStackValue":2},{"UAdd":{"result":0,"o
p1":1,"op2":2}},{"Mv":[1,2]},{"Mv":[2,0]},{"UAdd"
:{"result":3,"op1":3,"op2":5}},{"Le":{"result":6,
"op1":3,"op2":4}},{"JC":{"label":0,"cond":6}}]
```


Рис 4.4 – Код представлений як json файл

Bincode – це мінімальний бінарний формат розроблений командою servo. Він призначений для ефективного зберігання та перначі інформації. Дані записані у форматі Bincode гарантовано займають стільки ж місця, як і їх представлення і пам’яті, а у деяких випадках навіть менше. Так як Bincode – бінарний формат, то досить важко зрозуміти, що твориться у файлі, але саме він є рекомендованим для довгострокового використання зберігання коду.

Приклад коду з минулих рисунків у форматі Bincode наведений на рисунку 4.5.

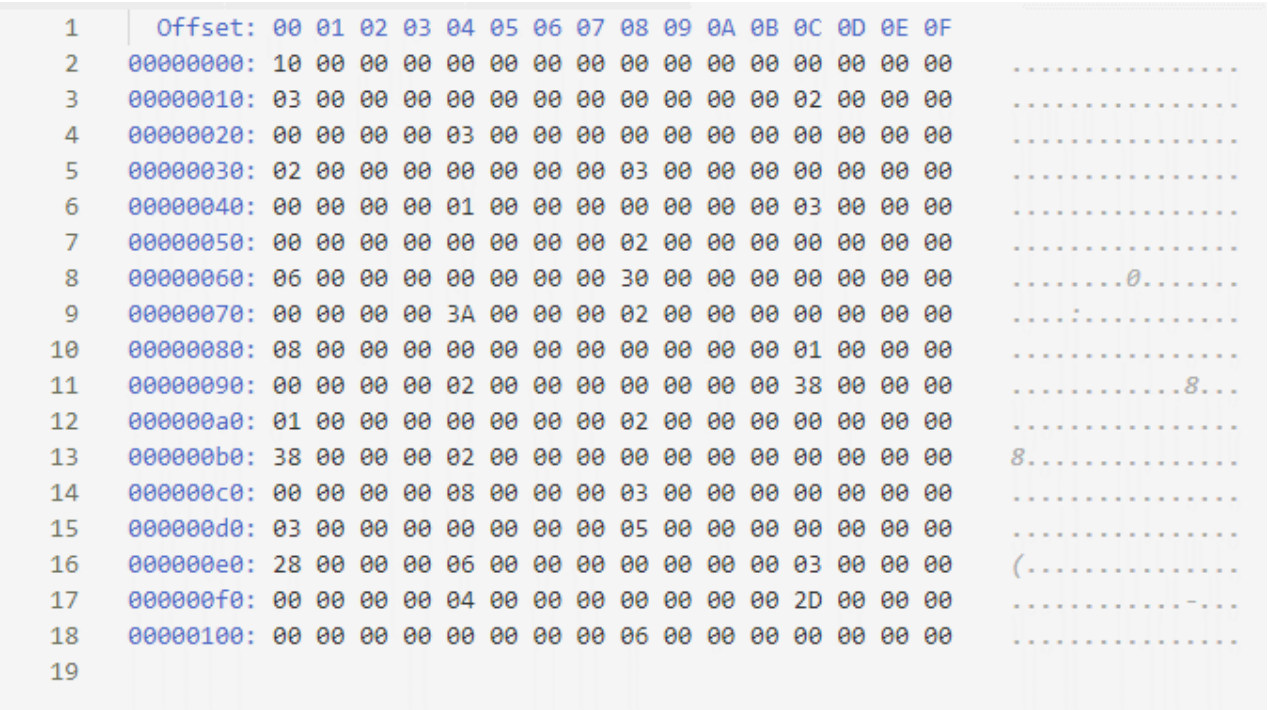


Рис 4.5 – Код у форматі Bincode

Як видно з рисунку 4.5, оскільки розмір кодових посилань в системі має як 8 байт, то bincode заповнений багатьма 0. Хоча це спрощує читання коду під час інтерпретації, дане архітектурне рішення значно збільшує об’єм bincode. На щастя, дана проблема дуже просто вирішується простим стисненням використовуючи вихідного файлу алгоритмом Deflate.

4.1.2 Виконання коду

Після задання коду одним із вищеописаних способів – можна виконувати код. Система має 2 основних режими виконання, що можна запустити відповідними функціями обєкту Vm: *headless* та *lite*.

В *headless* режимі система запускається без контексту виконання – необхідно. Все що необхідно – це надати пул констант для виконання. Цей режим є ідеальним для вбудованих сценаріїв, адже дозволяє виконувати будь-який код викликавши лише метод «interpret» на ньому. Приклад запуску системи в *headless* режимі наведено на рисунку 4.6.

```rust
let pool = ...;
let code = Code::from_model(code).ok_or(VmError::InvalidBytecode)?;
let mut vm = Vm::headless(pool);
code.interpret(&mut vm)
```

Рис 4.6 – Виконання коду в headless режимі.

Метод «interpret» повертає «Result<(), VmContextError>», тому у разі помилки виконання система надає всю необхідну інформацію для розуміння помилки у зручному вигляді, або дозволяє пропагувати помилку далі за допомогою «try» оператора.

У *headless* режимі зручно запускати програми із середньою на великою кількістю інструкцій. Описана в під-розділі 4.1.1 програма добре виконується в *headless* режимі. Результат виконання даної програми наведено на рисунку 4.7.

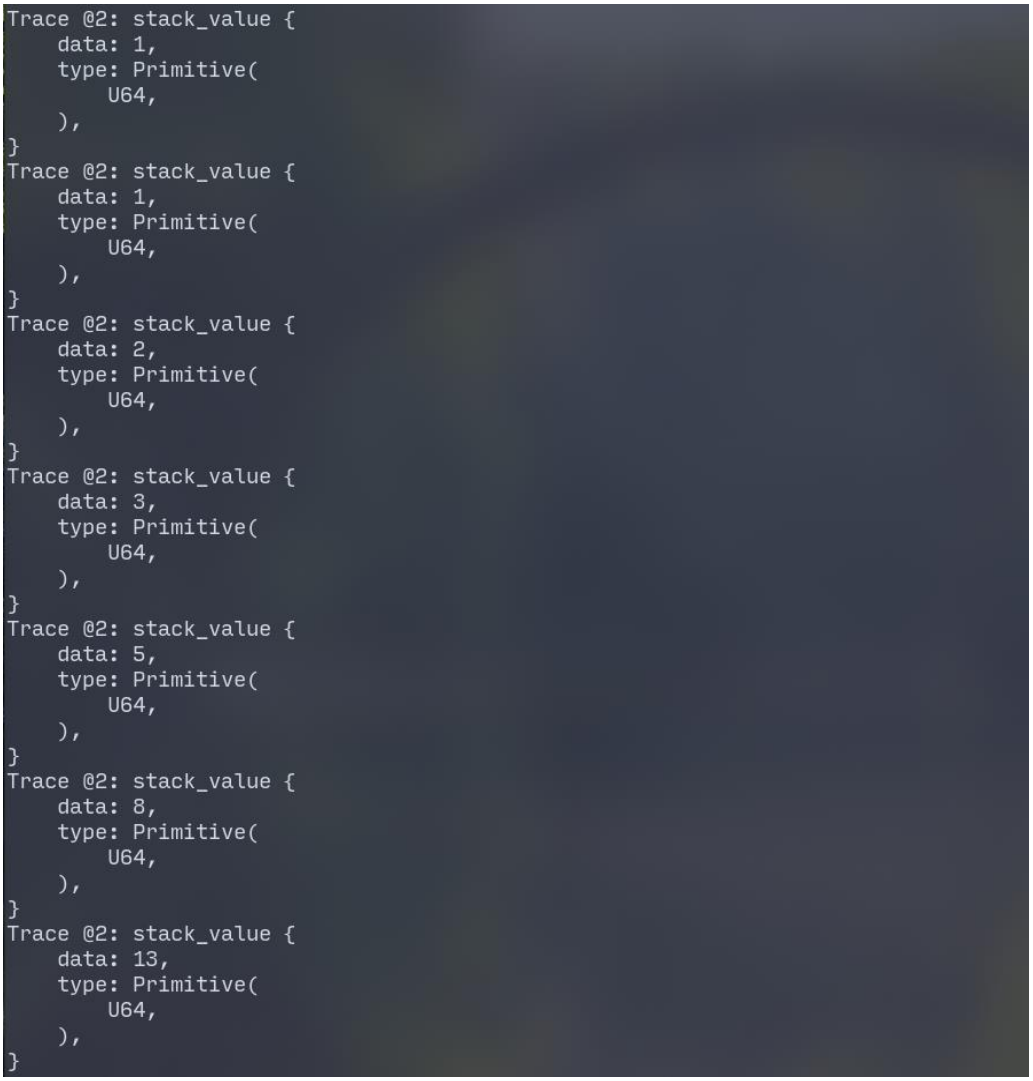


Рис 4.7 – Результат виконання коду в headless режимі

lite режим подібний до *headless*, але система не резервує ніякої пам’яті для виконання коду. Необхідна пам'ять буде виділена в момент виконання. Даний режим є ідеальним для виконання короткого набору інструкцій та дозволяє створювати велику кількість об’єктів Vm без використання великої кількості пам’яті. Так наприклад всім відомий hello world можна виконати на x64 Linux використавши лише 2 кб пам’яті. Приклад запуску системи для як-раз такої програми наведений на рисунку 4.8, а результат виконання – на рисунку 4.9.

```
fn main() -> Result<(), VmContextError> {
    let pool = ConstantPool::new(vec!["Hello wo
rld!".into()]);
    let code = Code::from_model(&[LdSS(p(0)), T
raceStackValue(s(0))]).unwrap();
    let mut vm = Vm::lite(pool);
```

Рис 4.8 – Запуск системи в *lite* режимі

```
Trace @0: stack_value {
    data: "Hello world!",
    type: Primitive(
        SStr,
    ),
}
```

Рис 4.9 – Результат виконання hello world в *lite* режимі

4.1.3 Аніліз коду за допомогою модулю *decoder*

Під час початкових стадій розробки системи було досить важко знайти точну проблемну точку. Саме тому, для внутрішнього використання, був розроблений модуль *decoder*, що представляє байт-код у текстовому форматі. Даний формат став дуже корисним у пошуку проблем, саме тому було прийняте рішення про додавання цього модуля до користувацького інтерфейсу системи. Приклад використання даного модуля наведений на рисунку 4.10

```
let decode = code.decode();
if !decode.is_full {
    panic!("Bad code")
} else {
    decode.print(true);
}
```

Рис 4.10 – Використання модулю *decoder*

Як бачимо з рисунку 4.6, байт-код надає асоційовану функцію «decode» що повертає DecodeResult. Поле «is_full» даної структури показує, чи вдалося повністю декодувати байт-код, при цьому якщо is_full = false, то при використанні декодування далі можна буде побачити точно, де відбулася помилка. Асоційовану функція «print» виводить на консоль результат декодування. Параметр вказує на те, чи необхідно показувати оригінальні байти з яких було зібране дане декодування. Приклад виконання даного фрагменту наведено на рисунку 4.11.

```
0   0x00                                                                U64Ld0
1   0x00                                                                U64Ld0
2   0x03000000000000000002000000000000                                  LdType $0 $2
19  0x03000000000000000002000000000000                                  LdType $0 $2
36  0x03000000000000000001000000000000                                  LdType $0 $1
53  0x03000000000000000002000000000000                                  LdType $0 $2
70  0x06                                                                LdFalse
71  0x00                                                                U64Ld0
72  0xfe020000000000000                                                 TraceStackValue @2
81  0x0a00000000000000001000000000000002000000000000                   UAdd @0 @1 @2
106 0x3c01000000000000002000000000000                                   Mv @1 @2
123 0x3c02000000000000000000000000000                                   Mv @2 @0
140 0x0a03000000000000003000000000000005000000000000                   UAdd @3 @3 @5
165 0x2a06000000000000003000000000000004000000000000                   Le @6 @3 @4
190 0x2f480000000000000060000000000000                                  JC *72 @6
```

Рис 4.11 – Результат виконання модулю Decoder

4.2 Порівняння роботи системи з іншими

В даному підрозділі порівнюється виконання створеної системи з рішеннями, що використовують інші способи управління пам’яттю. Прикладом системи з трасуванням було вибрано JVM, а прикладом системи з використанням підрахунку посилань було обрано CPython Vm.

Порівняння відбувається під час виконання 3 простих програм: *Hello world*, *Fibonacci*, *Array*.

Основні характеристики що будуть порівнюватися, їх опис та спосіб їх вимірювання наведено в таблиці 4.1.

Таблиця 4.1 – Характеристики для порівняння

Характеристика	Опис	Спосіб вимірювання
Branch Misses	Кількість промахів передбачення гілок. Показує наскільки код системи призначений для сучасних процесорів	valgrind cachegrind
L1D Misses	Кількість промахів кешу даних. Показує наскільки організація даних в системи підходить для сучасних процесорів	valgrind cachegrind
Time	Кількість циклів виконання процесора	perf stat -d
Max Resident Size	Максимальна кількість памяті, що була задіяна під час виконання	bin/time -v
Total allocated space	Загальна кількість памяті що була виділена у процесі виконання	valgrind memcheck
Allocs/Frees	Кількість викликів malloc/free програми. Рівна кількість цих чисел означає повне звільнення ресурсів системи після виконання	valgrind memcheck

Порівняння проводилося на ArchLinux x64 (Версія ядра 5.4) з процесором Amd ryzen 7 3700x @ 3.7 Ghz та 32ГБ DDR4 @ 2400Khz пам’яті, та на останній версії Rust nightly на момент проведення тестування.

Для більш справедливого порівняння JVM було запущенно в режимі інтерпретації, щоб не включати час та пам'ять витрачену на компіляцію коду в результат.

Програма тестування *Hello world* – просто виводить «hello world!» на консоль. Дана програма показує ресурси необхідні для ініціалізації системи та для виходу в користувацький код (до main функції). Цей тест є базовим для порівняння з іншими. Для даного тесту система була запущена в *lite* режимі. Всі інші тести виконуються в *headless* режимі. Результати наведені в таблиці 4.2

Таблиця 4.2 – Результати тестування *Hello world*

	ngvm	JVM	CPython Vm
Branch Misses	8.4%	7,5%	7,8%
L1D Misses	2.7%	2,5%	4,2%
Time	0,62 CPU	1,455 CPU	0,976 CPU
Max Resident Size	2440 кб	33 376 кб	8520 кб
Total allocated space	2 769 б	35 460 367 б	4 312 681 б
Allocs/Frees	28/28	29,238 / 4,829	4,736 / 4,598

Як бачимо, що для ініціалізації розроблена система використовує значно менше пам’яті, та обходиться лише 28 виділеннями пам’яті, в трохи більше як 2 кб. Також на відмінну, від JMV та CPython Vm, система повністю очищує всю виділену після використання пам'ять, та готова по повторної ініціалізації, що робить її ідеальної для основного призначення: вбудовування в інші проєкти.

На жаль, промахів передбачення гілок під ініціалізації було трохи більше ніж JVM та CPython Vm. Цей результат є очікуваним та пов'язаний з данно-орієнтованою архітектурою системи. Під час виконання ця кількість промахів значно зменшитися.

Програма тестування *Fibonacci* – рахує перші n (n=50) чисел Фібоначчі. Даний тест призначений для аналізу ефективності виконання 64 бітової арифметики в системах. Тут основною для порівняння є характеристика Time.

Результати виконання даного тесту наведені в таблиці 4.3.

Таблиця 4.3 – Результати тестування *Fibonacci*

	ngvm	JVM	CPython Vm
Branch Misses	6,7%	8,3%	7,8%
L1D Misses	0,9%	3,4%	4,1%
Time	0,861 CPU	1,450 CPU	0,977 CPU
Max Resident Size	2492 кб	33 388 кб	8520 CPU
Total allocated space	31 786 б	34 996 546 б	4 313 095 б
Allocs/Frees	860/860	29,301/ 4,914	4,736 / 4,598

Як бачимо з таблиці 4.3, на жаль, системі знадобилась досить велика кількість виділень пам’яті в порівнянні з Фібоначчі. Нажаль ця проблема виникла через обмеження перевірки позичань в мові Rust. В деяких ситуаціях довелося виділяти пам'ять, хоча це не є зовсім необхідним. Для усунення цих проблем можливі оптимізації деяких інструкцій з використанням unsafe коду та SmallVec типу. Також, як бачимо кількість Branch Misses значно зменшилась в порівнянні з конкурентами, хоча час виконання програми на CPU значно збільшився. Це можна пояснити постійними перевірками безпечності виконаних інструкції.

Програма тестування *Array* – виділяє постійно пам'ять у циклі, кожен раз відділяючи на 8 байт більше чим минулий. Цей тест показує те, як ефективно система використовує пам’ять батьківської системи. В даному тесті важливою характеристикою є Max Resident Size, Total allocated space Allocs та L1D Misses. Варто зазначити, що наприкінці виконання сам масив займає 390 кб. Будь-який надлишок виділеної пам’яті поверх цього не є необхідним. Результати тесту *Array* наведено в таблиці 4.4.

Таблиця 4.4 – Результати тестування *Array*

	ngvm	JVM	CPython Vm
Branch Misses	1,6%	1,6%	0,0%
L1D Misses	0,0%	0,0%	10,0%
Time	0,994 CPU	1,140 CPU	1,000 CPU
Max Resident Size	8 408 кб	1 109 532 кб	9620 кб
Total allocated space	10 816 247 б	41 519 060 б	10 004 096 267 б
Allocs/Frees	250 043 / 250 043	38,899 / 7,577	54,671 / 54,533

Як бачимо з таблиці 4.4: система значно краще працює за сценаріями постійного виділення великої кількості пам’яті, та не поступається трасуючій JVM. Хоча в даному випадку їй знадобилося значно більше викликів malloc/free пар в порівнянні з конкурентами. CPython vm мав аж 10% кеш промахів та значно більший об’єм виділеної пам’яті. Це сталося ймовірно, через реалізацію масивів як двозв’язних списків.

Загалом, система показує передбачувані результати. Вона використовує значно менше пам’яті для виконання та є більш оптимізованою під виконання на сучасному процесорі. Але дане тестування також показало очевидні слабкості системи, зокрема постійні перевірки безпеки та управління пам’яттю потребуюсь значно біль часу CPU ніж конкуренти в порівнянні з ініціалізацією, а також система виділяє досить багато пам’яті для простих арифметичних операцій. Отож, хоча результати тестування є досить хорошими, все ще існує досить багато місця реалізації для покращень.

ВИСНОВКИ ДО РОЗДІЛУ 4

В даному розділі представлені основні способи використання системи на прикладі виконання фрагменту коду. Показані основні способи задання коду для виконання, а також основні режими виконання системи. Також, був продемонстрований вбудований decoder байт-коду для знаходження помилок в коді.

У другому під-позділі, порівнюється виконання деяких фрагментів коду з JVM – системою виконання з використанням трасування, та CPython VM – системою виконання з використанням підрахунку посилань. Продемонстровано, що система показує очікувані результати за основними характеристиками що порівнюється, але також має деякі слабкості. Зокрема система використовує досить велику кількість малих виділень пам’яті при виконанні простих арифметичних операцій.

Загалом цей розділ показує, що описана в розділі 3 система та її архітектура є непоганим переосмисленням сталої архітектури віртуальної машини та добре пристосована до постійної переініціалізації та постійних викликів невеликих фрагментів коду. Саме для таких сценаріїв розроблялась і дана система.

ВИСНОВКИ

Даний дипломних проєкт представляє новий спосіб управління пам’яттю в системах виконання коду.

В першому розділу розглянуті існуючі системи виконання, їх архітектуру, основні переваги та недоліки їхньої реалізації.

В другому розділу була розглянута відносно нова мова програмування від Mozilla Foundation: Rust – високорівнева мова системного програмування, та показано чому саме вона є ідеальної для розробки нової системи виконання. В даному розділі, також, був проведений опис існуючих методів управління пам’яттю в системах виконання коду, та на противагу їм був представлений новий метод автоматичного управління пам’яттю оснований на циклах життя об’єктів системи. Враховуючи новий метод управління пам’яттю була розроблена архітектура системи та специфічних байт-код, що враховує як і особливості архітектури системи та сучасних процесорів, так і особливості управління пам’яттю в системі.

В 3 розділі представлені основні особливості модулів системи, що виникають під час реалізації системи виконання коду без використання методів збирання сміття на мові Rust. Був також продемонстрований перехід від байт-коду як основного способу задання коду, до об’єктної моделі. Це дозволило збільшити портативність системи, врахувати особливості батьківської системи під час збирання системи під нову архітектуру обладнання, а також значно спростити задання коду.

В четвертому розділу було продемонстровано основні способи використання системи, показано варіанти задання коду системи та продемонстровано виконання одного з таких кодів. Даних розділ також порівнює на деяких тестових даний основні характеристики виконання системи та показує, що вона справді підходить для розроблених сценаріїв підключення в інші проєкти.

Основним результатом даного дипломного проєкту є система виконання коду, що на відміну від своїх конкурентів реалізує автоматичний, але детермінований механізм управління пам’яттю оснований на циклі життя об’єктів. Завдяки цьому, вона прекрасно підходить для виконання у вбудованих сценаріях, на ввідмінну від існуючих рішень.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Lindholm T., Frank Y. Bracha G., Buckley A. The Java® VirtualMachine Specification, 2015. URL: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf
2. Bloom J. JVM Internals, 2013. URL: https://blog.jamesdbloom.com/JVMInternals.html
3. Java bytecode instruction listings. URL: https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings
4. Ierusalimschy R., de Figueiredo L., Celes W. The Implementation of Lua 5.0 URL: https://www.lua.org/doc/jucs05.pdf
5. Laurie D. Lua 5.2 Bytecode and Virtual Machine, 2013. URL: http://files.catwell.info/misc/mirror/lua-5.2-bytecode-vm-dirk-laurie/lua52vm.html
6. Rust, A language empowering everyoneto build reliable and efficient software. URL: https://www.rust-lang.org/
7. Klabnik S., Nichols C., The Rust Programming Language, 2018. URL: https://doc.rust-lang.org/book/
8. Crate std – Rust. URL: https://doc.rust-lang.org/std/
9. The Rust community’s crate registry. URL: https://crates.io/
10. Turon A. Abstraction without overhead: traits in Rust, 2015. URL: https://blog.rust-lang.org/2015/05/11/traits.html
11. Tracing garbage collection. URL: https://en.wikipedia.org/wiki/Tracing_garbage_collection
12. Reference counting. https://en.wikipedia.org/wiki/Reference_counting
13. Automatic Reference Counting. URL: https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html
14. IEEE 754-2008. Standard for Floating-Point Arithmetic. – 2008. – ISBN 978-0-7381-5752-8

15. Nystrom B. Crafting Interpreters, 2020. URL: http://www.craftinginterpreters.com
16. Zen 2, Microarchitectures – AMD. URL: https://en.wikichip.org/wiki/amd/microarchitectures/zen_2

ДОДАТОК А

СИСТЕМА ВИКОНАННЯ КОДУ БЕЗ ВИКОРИСТАННЯ МЕТОДІВ ЗБИРАННЯ СМІТТЯ

СХЕМА СТРУКТУРНА

Аркушів 1

Київ – 2020

ДП.6418. 04.000 Д1

Зм.	Арк.	№ докум.	Підпис	Дата
Розроб.		Пархоменко Д. В.		
Перевір.		Сімоненко А.В.		
Н. контр.		Сімоненко В.П.		
Затверд.		Стіренко С.Г.		

ДП.6418. 04.000 Д1

Система виконання коду без використання методів збирання сміття
Структурна схема

Літ.	Маса	Масштаб

Арк. 1	Аркушів 1

Дипломний проект

КПІ ФІОТ
кафедра ОТ
гр. ІП-64

ДОДАТОК Б

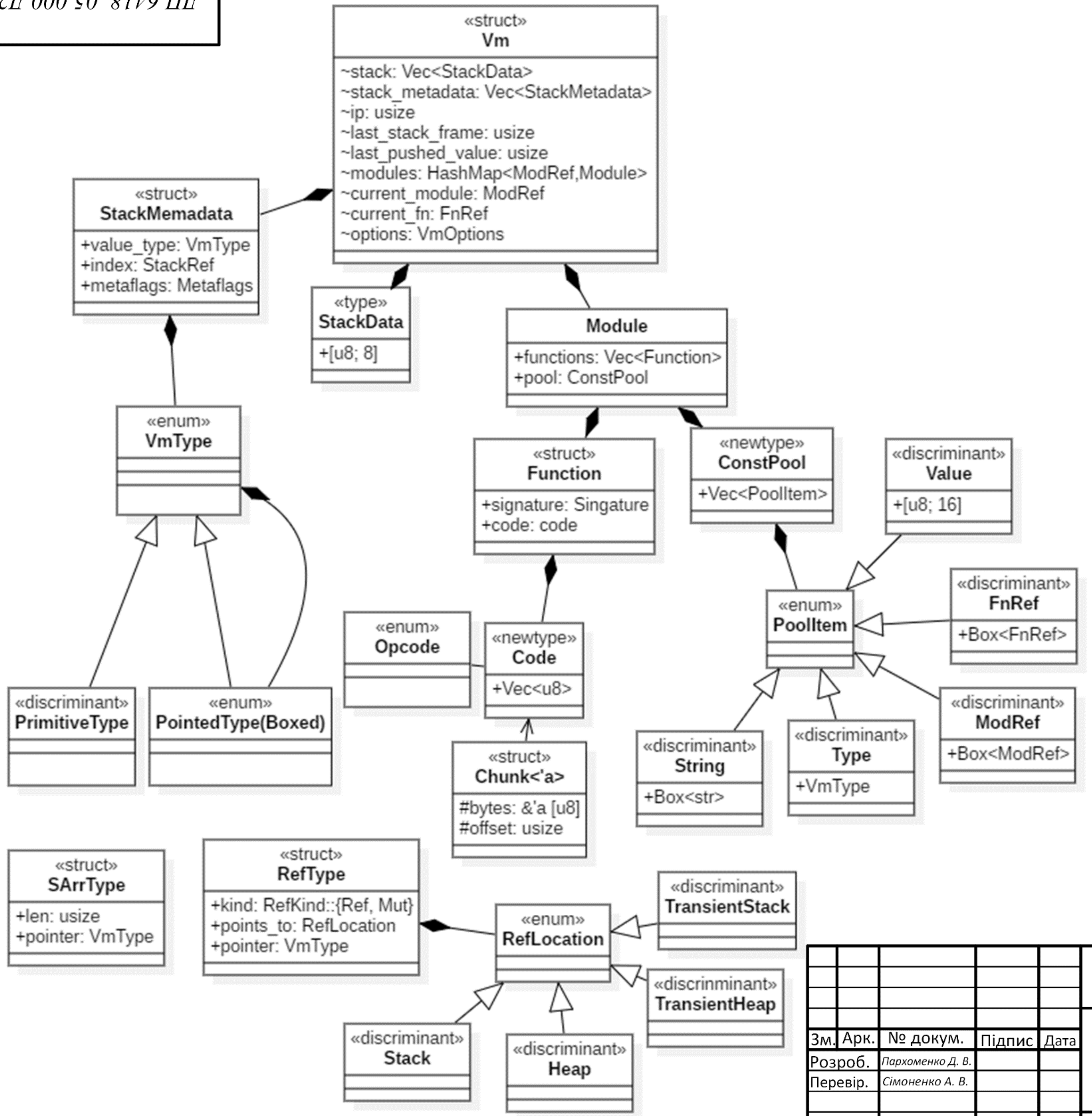
СИСТЕМА ВИКОНАННЯ КОДУ БЕЗ ВИКОРИСТАННЯ МЕТОДІВ ЗБИРАННЯ СМІТТЯ

СХЕМА ФУНКЦІОНАЛЬНА

Аркушів 1

Київ – 2020 року

ДП.6418. 05.000 Д2



Зм.	Арк.	№ докум.	Підпис	Дата
Розроб.		Пархоменко Д. В.		
Перевір.		Сімоненко А. В.		
Н. контр.		Сімоненко В.П.		
Затверд.		Стіренко С.Г.		

ДП.6418.05.000 Д2

Система виконання коду без використання методів збирання сміття

Функціональна схема

Літ.	Маса	Масштаб

Арк. 1 | Аркушів 1

Дипломний проект

КПІ ФІОТ
кафедра ОТ
гр. ІП-64

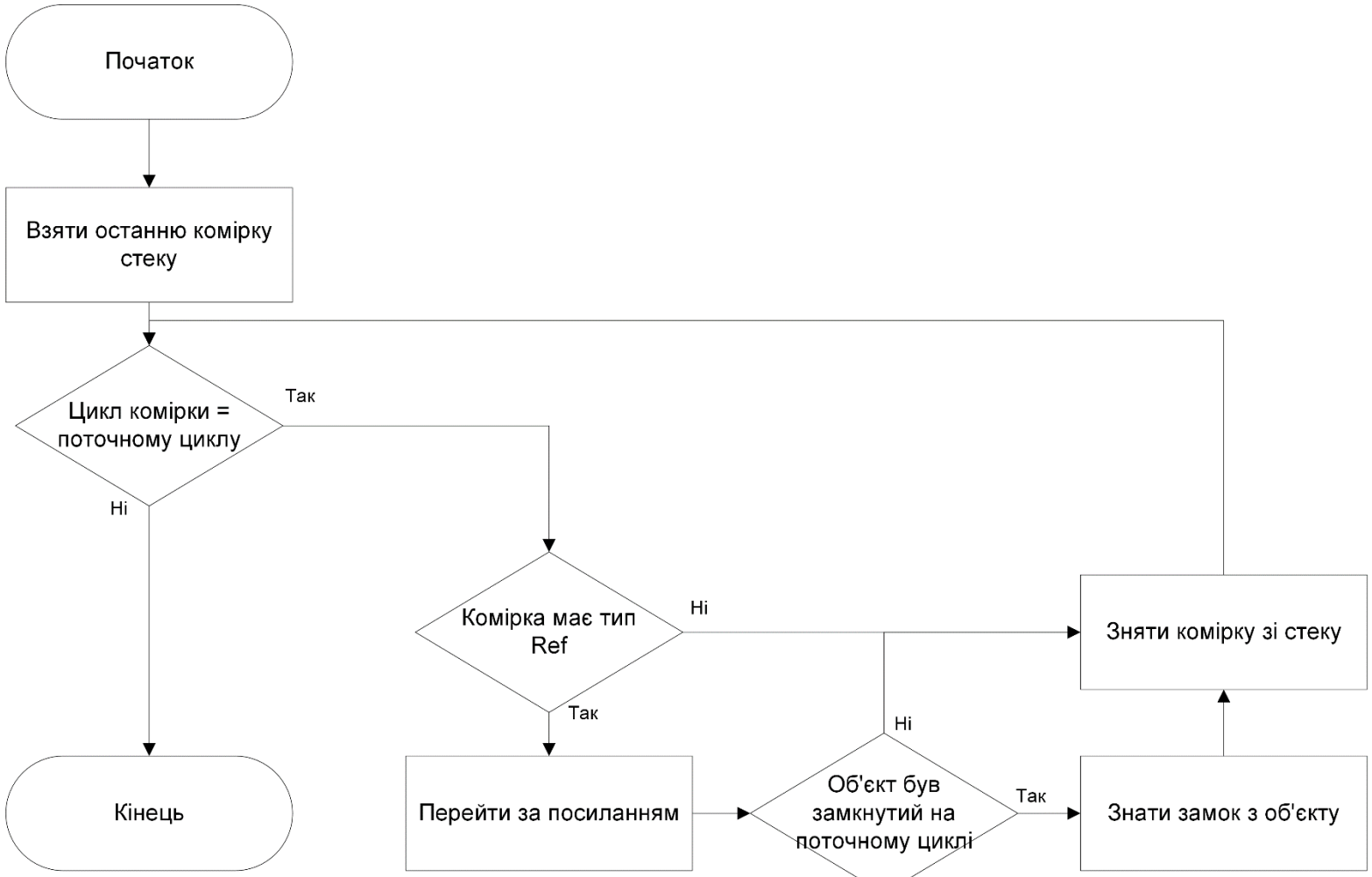
ДОДАТОК В

СИСТЕМА ВИКОНАННЯ КОДУ БЕЗ ВИКОРИСТАННЯ МЕТОДІВ ЗБИРАННЯ СМІТТЯ

БЛОК СХЕМА АЛГОРИТМУ

Аркушів 1

Київ – 2020



					ДП.6418. 06.000 Д3	
Зм.	Арк.	№ докум.	Підпис	Дата	Система виконання коду без використання методів збирання сміття Блок схема алгоритму	Літ. / Маса / Масштаб
Розроб.		Пархоменко Д. В.				
Перевір.		Сімоненко А. В.				Арк. 1 / Аркушів 1
Н. контр.		Сімоненко В.П.			Дипломний проект	КПІ ФІОТ кафедра ОТ гр. ІП-64
Затверд.		Стіренко С.Г.				